REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time reviewing instructions, searching existing data sources gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE March 1995	3. REPORT TYPE AND DATES COVERED Master's Thesis

4. TITLE AND SUBTITLE
A NETWORKED VIRTUAL ENVIRONMENT FOR SHIPBOARD TRAINING (U)

5. FUNDING NUMBERS

6. AUTHOR(S)
King, Tony Edward
McDowell, Perry Lewis

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/ MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING/ MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the United States Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Operational shipboard environments are characterized by uncertainty, short time constraints, stress, multiple sources of information and teamwork. However, most naval training ignores the fundamental three-dimensional and team natures of both the environment and human perception. The problem addressed by this research is to improve the quality and reduce the expense of training for naval personnel.

Our belief is that this problem can be solved by training sailors in a Virtual Environment for Training (VET). Virtual environment trainers are ideally suited to address the above shortcomings and provide better and more intuitive training at a lower cost than current methods. However, such an environment has not been proven theoretically possible. Our approach is to create such an environment, which can then be evaluated for its training effectiveness.

This thesis proves the feasibility of a virtual environment to solve the Navy's training problem. We built a real-time, distributed, interactive shipboard environment for training. It consists of a three-dimensional ship model, which consists of objects containing over 22,000 polygons; an application program, which can render this model with average frame rates of fifteen to twenty frames per second; and networking code, which can include a theoretically unlimited number of participants, although performance suffers with greater than ten participants. The participants can interact in the same virtual ship to combat several likely casualties, including a fuel oil leak, main space fire, and steam rupture.

14. SUBJECT TERMS
Walkthrough, Virtual Environment, Training, Navy, Network, Simulation, 3-D Modeling

15. NUMBER OF PAGES

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

A NETWORKED VIRTUAL ENVIRONMENT FOR SHIPBOARD TRAINING

by

Perry L. McDowell
Lieutenant, United States Navy
B.S., United States Naval Academy, 1988
and
Tony E. King
Lieutenant Commander, United States Navy
B.S., United States Naval Academy, 1982

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE
from the
NAVAL POSTGRADUATE SCHOOL

March 1995

Authors:
_____
Perry Lewis McDowell

_____
Tony Edward King

Approved By:
_____
David R. Pratt, Co-Thesis Advisor

_____
Michael J. Zyda, Co-Thesis Advisor

_____
Ted Lewis, Chairman,
Department of Computer Science

iii

ABSTRACT

Operational shipboard environments are characterized by uncertainty, short time constraints, stress, multiple sources of information and teamwork. However, most naval training ignores the fundamental three-dimensional and team natures of both the environment and human perception. The problem addressed by this research is to improve the quality and reduce the expense of training for naval personnel.

Our belief is that this problem can be solved by training sailors in a Virtual Environment for Training (VET). Virtual environment trainers are ideally suited to address the above shortcomings and provide better and more intuitive training at a lower cost than current methods. However, such an environment has not been proven theoretically possible. Our approach is to create such an environment, which can then be evaluated for its training effectiveness.

This thesis proves the feasibility of a virtual environment to solve the Navy's training problem. We built a real-time, distributed, interactive shipboard environment for training. It consists of a three-dimensional ship model, which consists of objects containing over 22,000 polygons; an application program, which can render this model with average frame rates of fifteen to twenty frames per second; and networking code, which can include a theoretically unlimited number of participants, although performance suffers with greater than ten participants. The participants can interact in the same virtual ship to combat several likely casualties, including a fuel oil leak, main space fire, and steam rupture.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I. INTRODUCTION

A. MOTIVATION

1.   The Navy's Training Dilemma

Operational shipboard environments are characterized by uncertainty, short time constraints, stress, multiple sources of information and teamwork. Often, crew members are required to make life and death decisions based upon the information gathered in such an abnormal environment. However, most naval training ignores the fundamental three-dimensional and team natures of both the environment and human perception.

Most Navy training for shipboard personnel occurs on the ships themselves. However, on ships, it is impossible to produce actual casualties for the crew to fight for training. For example, it is unrealistic to knock a hole in the side of a ship to create a flooding drill to train damage control personnel. To account for this problem, the Navy has two solutions.

The first is to simulate the casualty aboard the actual ship. This is done by using two-dimensional props during drills, such as a piece of paper with a hole drawn on it to simulate a loss of hull integrity, a chem-light to simulate fire or flapping blankets to simulate a steam rupture. While this training can be done anywhere the ship goes, its realism is sorely lacking.

The other method is to simulate the ship and have the actual casualty. This is what is done at the Navy's shore based training simulators. These physical mock-ups look like a ship and are designed to have actual casualties, such as real fires, flooding or engineering casualties for the sailors to fight. While the sailors get much better training at the simulators, they too have serious drawbacks. They are very expensive to build and require many man-hours to operate and maintain. For these reasons, they are relatively scarce, especially compared to the number of ships needing to use them. Because of this, it is often difficult for ships to schedule training in these mock-ups due to the trainer's lack

of availability. Also, a ship's schedule is very unpredictable, and ships often miss their assigned training slots due to unexpectedly getting underway.

The reason that the Navy has drills at sea and at these expensive simulators rather than having only classroom training is that humans have a remarkable capacity to process spatial information. Lessons which are learned actively in a realistic environment are more likely to be retained by the trainee, and more importantly, more likely to be applied during stressful situations than those learned passively in the classroom. The Navy needs a training device which combines the realism of the physical trainers, the low cost of classroom training and the portability to be taken with the ships when they leave port.

2. Why Virtual Environments

This need can be met with the use of today's low-cost, high-speed computers using current virtual environment (VE) technology. VE technology developments over the past decade have produced the ability to synthesize large scale, three-dimensional models, such as a Navy ship, and produce a real time, interactive simulation. Advances in network systems allow these graphical simulations to communicate, enabling a large number of people to interact in the same virtual environment.

Although these virtual environments can be created, their capacity to solve the Navy's training problem must be demonstrated before building them makes sense. Since virtual displays surround users with three-dimensional stimuli, personnel in virtual environments feel a sense of "presence," that is, that they are actually inhabiting a place instead of looking at a picture of it. This aspect is illustrated in Figure 1. By being immersed in the environment, they naturally interact and familiarize themselves with it. When personnel are immersed in a virtual environment, they do not have to transform their natural perceptions to fit into the environment as with a traditional, two-dimensional display. Therefore, personnel can interact within this virtual environment using the same natural semantics that they use when interacting with the physical world, which means that they are more likely to take lessons they learn there into the real world. [HITL94]
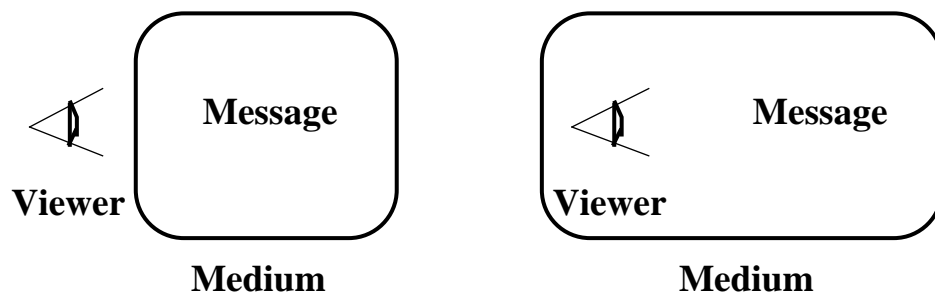
Figure 1: Comparison of Traditional and Virtual Interfaces.
From [HITL94]

VE technology is ideally suited to safely prepare Navy personnel for the dangerous environments they encounter on Navy ships. Virtual environments can produce a training environment far more realistic than the two dimensional simulations currently in use aboard ships, at a cost far below that of shore based trainers, and they are able to be used by the ship wherever it goes.

It is the goal of this thesis to produce a prototype of an interactive, real-time, networked, virtual training platform which can be installed on board Navy ships. This prototype can then be evaluated for its effectiveness in training personnel.

B. BACKGROUND

Modeling a Navy ship and designing an interactive, real-time networked virtual environment for moving through that ship has never been done. The closest work of this nature is architectural walkthroughs of large scale models. However, relatively few researchers have completed such systems because of the inherent problems associated with them.

It has been estimated that visual reality as envisioned in a computer graphics image consists of eighty million polygons per picture [CATM84]. Couple this number with the fact that in order to provide a real time interactive experience, ten frames per second is a conservative minimum acceptable frame rate. To be able to render a scene of that

3

complexity at that speed, a computer would have to be capable of rendering eight hundred million polygons per second. Even the most optimistic estimates do not foresee computers reaching this speed any time in the near future.

However, it is not necessary to reach that level before effective virtual environments can be created. Complete reality may be eight hundred million polygons per second, but very realistic simulations can be rendered using only millions to tens of millions of polygons per second. These speeds are still far beyond the rendering capabilities of today's workstations, but there exist methods which make it possible to produce effective walkthroughs using this equipment. Doing so requires high priced graphics workstations, high performance rendering software and efficient database management algorithms. The list of those who have overcome the obstacles and built large scale, interactive virtual environments include Airey, Rohlf and Brooks at the University of North Carolina (UNC), Chapel Hill, and Teller, Sequin and Funkhouser at the University of California (UCB), Berkeley. The methods devised to walkthrough and interact with the virtual ship trainer are largely predicated on the work of these individuals. [AIRE90A] [FUNK94]

Since 1986, a dedicated team of computer scientists from UNC, led by Fred Brooks, has consistently been at the leading edge of architectural walkthrough development. They have built a succession of interactive computer graphics systems which enable a viewer to experience an architectural design by simulating a walkthrough of a model. Rather than using off the shelf systems, UNC uses graphics hardware architectures they have built in-house; the first was PixelPlanesI, which eventually evolved to PixelPlanes5 and its successor, their current architecture, PixelFlow. The increase in speed has been dramatic; in 1986, PixelPlanesI rendered an 8,000 polygon model with flat shaded polygon walls at a frame rate of one frame every three seconds. Recently, PixelPlanes5 has achieved update rates of fifty frames per second on models of thirty-thousand polygons. Additionally, they have added many features to enhance the sense of realism in the virtual environment. These include:

- Illumination using radiosity calculations and Gouraud Shading.

- Texturing of wall, floors and furnishings.
- Spatialized sound.
- Use of stereo head-mounted displays.
- Tracking user movement in a twelve foot by ten foot space and translating it to the virtual environment.
- Ability to pick up objects, move furniture and otherwise interact with the environment using three-dimensional mouse. [BROO92]

The group under Carlo Sequin at UCB has also been active in architectural walkthroughs. They have designed an interactive visualization of a large scale architectural model, consisting of over 1.4 million polygons. Originally stimulated by Airey's work at UNC in model partitioning, they have experienced tremendous success in obtaining near real-time simulations using model partitioning and level of detail algorithms [FUNK94]. Their work foreshadows the future developments which will make large-scale walkthroughs a reality.

C.   PROBLEMS IN BUILDING A VIRTUAL SHIP

There are several problems associated with creating a networked, interactive virtual ship. Modeling the ship requires meticulous attention to detail and requires a great deal of time. The interactive mechanisms, such as navigating through the ship, moving objects, casualty scripts and interface displays need to be well designed in order to provide a sense of realism to the user. Finally, to provide a networked environment to instill team training concepts, network protocol and packet design issues needed to be decided and implemented. These problems and others are discussed in greater detail below.

1.   Modeling

The problems with high polygon counts mentioned above are even more severe in models of ships. To produce the same degree of realism, a model of a single compartment in a ship requires over eight times the number of polygons as an entire house and almost six times as many as an entire church [MINE94]. This means that the model must be designed extremely well to allow it to be visualized in real time.

Naval architects have recently discovered and begun to use computer aided design (CAD) tools. These tools are now used in all aspects of ship construction, from the early concept stage, through the final design, into the actual building of the ship and continuing into the modifications during the ship's thirty year life cycle. It would be ideal to convert one of these CAD models of a ship to a polygonal database structure which could be rendered by three-dimensional visualization software. Had that been possible, it would be a much quicker process to create a model which provided accurate scaled details of all the spaces and equipment. However, several stumbling blocks were encountered while attempting to reach this objective.

The first stumbling block was trouble in obtaining CAD data of a Navy ship. Naval Sea Systems Command did not provide CAD data in time to be used for this thesis. The data was not received until January, 1995, when it was too late to attempt to incorporate it into this thesis.

In addition, once CAD data was received, it was in a form unique to a shipbuilding company. The only method to convert the CAD data into a form which can be easily visualized was to write software to do this. This presented several difficulties. Architectural models were originally created for the generation of blueprints. The drafting packages used to prepare them were not written anticipating that their results would be used to create data for interactive walkthrough systems. Consequently, when converting CAD data, many objects are not modeled as closed polygons, many lines do not connect, polygons are drawn with no consistent orientation and many coplanar polygons coexisted. Because of these problems, all systems which convert CAD data into visualization data require human intervention at some point, normally to correct problems such as back facing or coplanar polygons. [ALSP92]

2. Real-time Rendering

As previously discussed, when a model is too large to be normally visualized at an acceptable speed, methods must be found to render it at real-time frame rates of ten to

twenty frames per second. These methods attempt to discover how to convert this large problem into several smaller problems, only a few of which must be solved to render the scene. Doing so reduces the amount which must be stored in memory and the computational load on the system to a level which today's machines are capable of handling. While this sounds simple in theory, it is extremely difficult to do this in a manner which maintains the fidelity of the simulation.

3. Design Implementation

Most work in interactive visualization involves vehicle simulators rather than architectural walkthroughs. Although there are many similarities, two major differences exist which pose new problems to the designer of the walkthrough. First, the motion or viewpoint control in a vehicle simulation is modeled after the vehicular constraints. The vehicles usually do not change direction suddenly or spin around, whereas in a walkthrough, the human user may perform these type of movements. Secondly, in vehicle simulators there is no need to detail objects precisely, since viewing usually occurs at a distance. On the other hand, architectural walkthroughs must be created with great care to detail objects precisely, since the viewer inspects objects at closer ranges.

D. SOLUTIONS

In order to reach the final objective of providing a networked, interactive virtual environment of the interior of a ship and also providing the tools necessary to interact with the environment in a natural way, several of the problems discussed previously were overcome. This section presents the solutions to those different problem areas.

1. Modeling

Modeling three-dimensional compartments, piping systems, pumps, and other equipment on board a Navy ship using a software modeling tool takes a tremendous number of man-hours to accurately represent the object for realistic visualization. Because of the problems using CAD data mentioned previously, all compartments, piping, pumps

and objects in the interior of the ship were modeled directly using MultiGen modeling software. Fortunately, the modeling began with a framework; Advanced Marine Enterprises, a contractor which performs work for NAVSEA, released a model they created as part of their contracting work. The model was of the *Antares*, a roll-on/roll-off ship under design. Even starting with this framework, many painstaking man-hours were required to build compartments, size piping, and produce equipment to create a realistic ship environment.

The final ship model contains only a fraction of the equipment found on board an actual ship. It contains a partially outfitted Engine Room, Combat Information Center (CIC), Damage Control Central (DCC), Operations Office, Hull Technician Workshop, Radar Room and passages between them. Other than CIC, which is based upon the CIC of an Aegis class cruiser, these compartments are not modeled after another ship design, but instead are a generic representation of what is normally found on ships.

2. Real Time Rendering

a. *Model Size*

To effectively evaluate the real-time rendering algorithms used, the polygon count of the model needs to be sufficiently high to slow down the frame rate. The final model contains almost twenty-three thousand polygons, which without performance tuning reduces the frame rate to less than three frames per second in some areas of the ship. If the model had been larger it would have been a better test of the performance enhancements, but the current model is sufficiently large to test the effectiveness of the algorithms.

b. *Performance*

Another extremely important goal of this system is to provide high frame rates to maintain the real-time feel of interactive visualization. If frame rates are too slow or too variable, the illusion of being present in a virtual environment is diminished significantly. If there are long time responses to input devices, referred to as high latency,

the feeling of presence is also degraded. Therefore, the goal is to maintain a frame rate of at least ten frames per second, which is the accepted standard for walkthrough environments, and maintain interactive response times less than a tenth of a second in order to obtain a sense of realism [NATI94]. To achieve this, an algorithm was designed relying on a hierarchical visual database which uses several methods to increase performance, including levels of detail, potentially visible sets, and instancing.

c. *Natural Navigation and Interaction*

Ideally, a user should be able to actually walk and feel the environment of the interior of the ship as he moves through it. Physical motion powerfully aids the illusion of presence, and actual walking enables one to feel kinesthetically how large spaces are. When opening a door or grabbing a fire hose, the user should be able to feel the object. Unfortunately, state of the art interfaces are still far from achieving this. Even though this ideal is currently impossible, the application strives to maintain a feeling of actually being in a ship. A physically based walking algorithm using a mouse is used to traverse the ship. Also, the user can use the mouse to manipulate objects, such as doors, nozzles and valves.

To increase the feeling of realism, several collision detection routines are employed. These are used to maintain the user's height of eye at a constant level above the deck and prevent the user from moving through objects. The collision routines also allow the user to go up and down ladders, open doors, pick up objects such as a fire hose, open the nozzle to spray water onto a fire, and open and close valves.

d. *Immersed Environment*

The more immersed an individual is within the virtual environment, the greater the illusion of actually being there is. One of today's best devices for creating immersive environments is the head-mounted display (HMD). An HMD provides the user with a wide field of view of the virtual environment while preventing the physical environment around them from being viewed. With the aid of a tracking device, the user's natural head motions allows him to look around the virtual environment. However, many

people do not like to wear HMD's for long periods of time. Therefore, two versions of this application were designed, one using an HMD and the other using a standard desktop monitor.

3. Networked Simulation System

The goal of networking the simulation is to allow participants on different workstations to enter the same ship, move through it, and interact cooperatively. This will ultimately lead to a team trainer where individuals of a damage control team can practice fighting a fire together while being networked on different computers. Using the Distributed Integration Simulation (DIS) network protocol, non-articulated humans have successfully been networked into the ship. Display data such as doors, fires, fire hoses and smoke also have been networked, so that the display is the same for each of the individuals participating in the simulation.

4. Training /Tutoring Tools

a. Path Planning

To aid in familiarization training, the simulation provides the user a tool which takes him along the path from any place on the ship to one of a limited number of destinations. By simply selecting a location from a menu, the user is translated to the location at walking speed, which gives him the benefit of viewing the path at normal speed.

b. Hyper-Text Displays

The user is able to select any object on the ship with an input device and retrieve information embedded into the visual database. This allows the user to discover the name and function of objects which are unfamiliar to him.

E. ORGANIZATION

The remainder of the thesis is broken down as follows:

- A system overview is presented in Chapter II. It includes a discussion of the hardware and software systems used. This chapter also provides an overview of

the design strategies employed to build the ship virtual environment trainer.

- Chapter III discusses the process used to create the model.

- Chapter IV introduces the user interface used in this simulation.

- Chapter V describes the path planning algorithm and how it is implemented.

- Chapter VI goes into detail describing how collision detection is used as the user moves through the database and picks objects.

- Chapter VII discusses using potentially visible sets to improve performance.

- Chapter VIII describes how environmental conditions are used to enhance the realism of the simulation and how the various casualty scenarios make use of them.

- Chapter IX discusses how users at different workstations are networked together to interact in the same virtual environment.

- Chapter X provides a final discussion of the results of this thesis and describes follow-on work to be accomplished.

II. OVERVIEW

This chapter summarizes the hardware, software and implementation process used to design the virtual ship. It also introduces the IRIS Performer tool kit and many of its features which this thesis will refer to throughout.

A. HARDWARE

The application was evaluated on three different types of graphics machines. The highest level machine used to evaluate performance is a Silicon Graphics Onyx Reality Engine 2 machine. The Reality Engine 2 incorporates a multiprocessing architecture, PowerPath2, to combine up to 24 parallel processors based upon the Mips R4400 RISC CPU, which operates at 150 MHz. I/O bandwidth is rated at 1.2 GB/second to and from memory, with support for the VME64 64-bit bus, operating at 50 MB/second. The Reality Engine 2 is rated at 2M flat triangles/second and 900K textured. [NATI94]

The middle machine used to evaluate the simulation is still a high level machine by most standards, the Silicon Graphics Power Series Reality Engine I. This machine has four R3000 40MHz processors, a single RM4 board and an integral SCSI controller. It is rated at 1.1M flat triangles/sec and 160M textured pixels. Currently, this machine is no longer being manufactured, but a machine of similar graphics capability is expected to be available by the end of 1995 for $29, 000. This is the machine which the authors' propose placing on Naval vessels to run this simulation.

The low end machine used in the Silicon Graphics Indigo 2 Extreme. It is a single processor machine, having only one R4400 CPU operating at 200 MHz. It is rated at 455K flat triangles/sec and 150K polygons/sec. It's biggest drawback is that it lacks the hardware and memory to handle textures, so it does not apply textures to polygons.

B. SOFTWARE

1. Rendering Software

To support the objective of real time graphics simulation, IRIS Performer was chosen to create the rendering software. Performer is an application programming interface whose architecture is designed to support high performance, multi-processed graphics applications, especially visual simulations, virtual reality and real-time, three-dimensional graphics. Performer is based upon hierarchical database, an example of which is shown in Figure 2. Performer culls the database so that only potentially visible geometry is sent to the graphics pipeline, provides fast intersection testing through database traversals, and performs level of detail management of objects. Most importantly, the cull, draw and application processes can run in parallel, greatly increasing the speed of the program. [ROHL94]
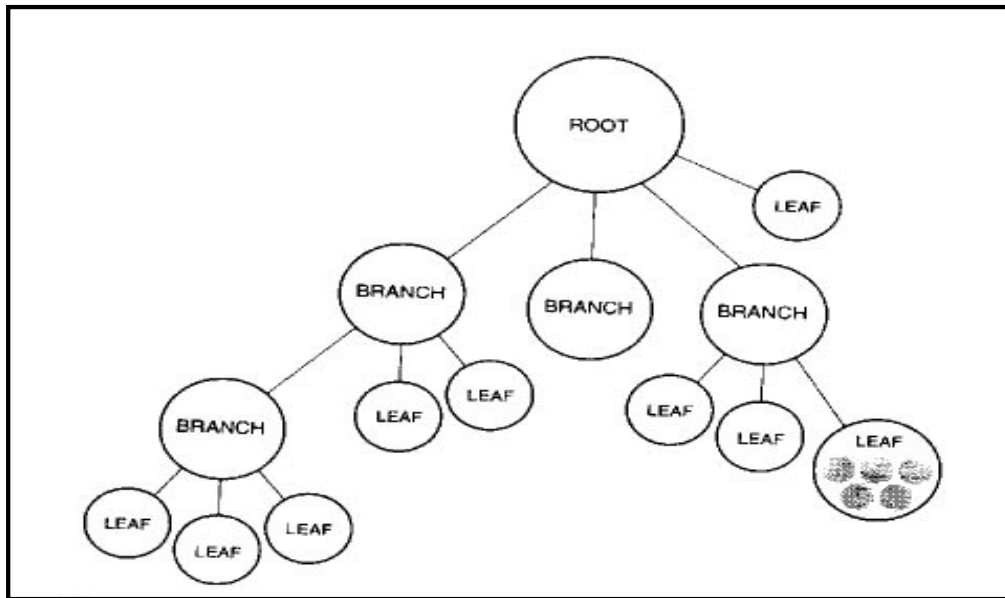
Figure 2: Example Scene Hierarchy

2. Modeling Software

To model the ship, the MultiGen Modeling Tool produced by Software Systems was used. It is a comprehensive format that can represent nearly all of IRIS Performer's

advanced concepts, including object hierarchy, instancing, level of detailing, light point specification, texture mapping and material property specification. It is discussed in more detail in Chapter III.

C. WALKTHROUGH OVERVIEW

1.   Rendering the scene

The simulation displays a three dimensional model created for specifically for it; the modeling process used to create the model is discussed in detail in Chapter III. To display the model, the application must provide the rendering software with the user's viewpoint. The viewer's field of view and far and near clipping planes also need to be specified. This information allows the rendering software to calculate a viewing frustum, which is the volume visible to the user. An example of a viewing frustum is displayed in Figure 3. The frustum is defined by the horizontal and vertical fields of view, which are the arcs which define the user's vision. The frustum is also defined by the near and far clipping planes; if something is closer than the near clipping plane, it is not displayed because it is too close for a human to focus on. Likewise, objects farther than the far clipping are too small to be seen, so they are not displayed.

The method in which the display is formulated is performed by a cull traversal. The cull traversal traverses the hierarchical bounding volumes provided by the scene graph, an example of which is shown in Figure 2. These bounding volumes define the area that an object and all its children inhabit. These volumes can be defined as several different type of shapes, although normally they are defined as cubes, spheres, or cylinders to simplify intersection calculations. During the cull traversal, the bounding volume of each node is compared against the viewing frustum. The action taken by the traversal depends upon the bounding volume test as follows:

- If the bounding volume is completely outside the frustum, entire branch is pruned and traversal continues without traversing any of the node's children.
- If the bounding volume is completely inside the frustum, entire branch is included and traversal continues.
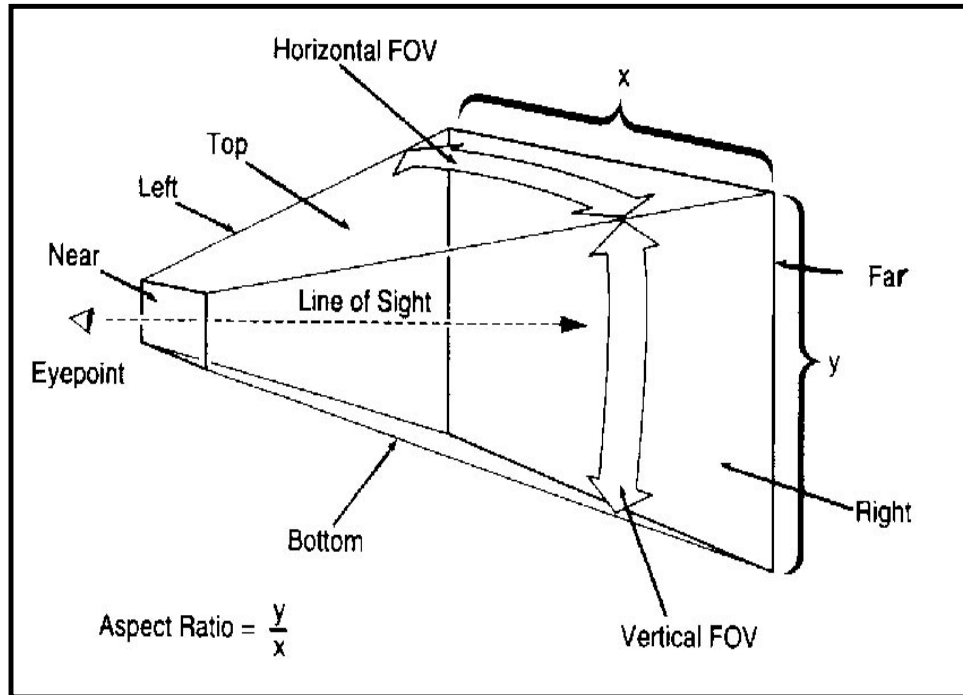
Figure 3: View Frustum.

From [SGI94], used with permission.

- If the bounding volume is partially inside the frustum, continue testing and traversing down the children of the node.

The final output of the cull traversal is geometry and graphics state information which is sent to the graphics hardware. [ROHL94]

Another traversal which takes place prior to each frame is the draw traversal. The draw traversal simply traverses the display list generated by the cull traversal and determines which of the objects in the display list are actually visible. This takes into account that an object might be occluded from the user's view by another object, as the disc in Figure 4 is hidden from the user's view by the wall. After determining the visible polygons, the draw traversal sends commands to the graphics subsystem. The draw and cull traversals automatically happen prior to each frame.

The final traversal which occurs before each frame, the intersection traversal, is performed only if invoked by the application. Intersections are based solely on a set of line
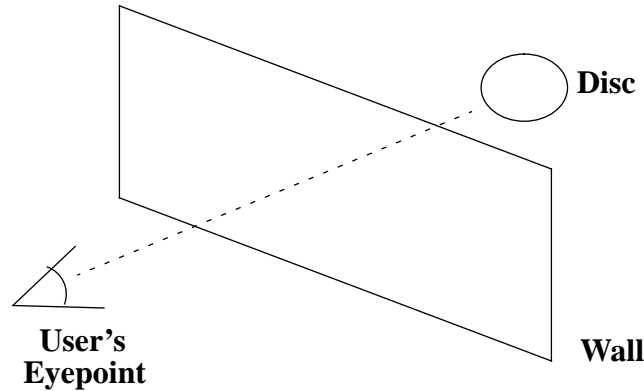
Figure 4: Example of Occluded Object

segments and intersection masks. When traversing the scene graph, a set of line segments is tested against the bounding volumes of the scene hierarchy. If an intersection occurs, information about the object collided with is returned. It is possible to discriminate between the types of objects collided with; Chapter VI describes the applications of this feature.

So far, three different processes or traversals have been discussed - cull, draw and intersection. The application process is the final process taking place each frame. The application process reads input from control devices, simulates the dynamics for motion, updates the visual database and performs network reads and writes. Shared memory is used to allow the four processes to use the same data. The interaction between shared memory and the four processes running in two windows or pipelines is illustrated in Figure 5.

By updating shared memory through repeated calls to the application, cull, draw and intersection processes, the scene display is refreshed each frame. With a single processor and a large visual database, the frame rate will drop very low. With multiple CPU's, IRIS Performer can distribute the application, intersection, culling and draw processes between different CPU's. This ability to multi-process greatly increases the speed at which each rendering cycle occurs. This capability is a major reason that IRIS Performer was chosen over other three-dimensional visualizer tool kits.

Performer uses its own coordinate system, displayed in Figure 6, which is slightly different than most others used in graphics. A view point is specified by the X, Y

17
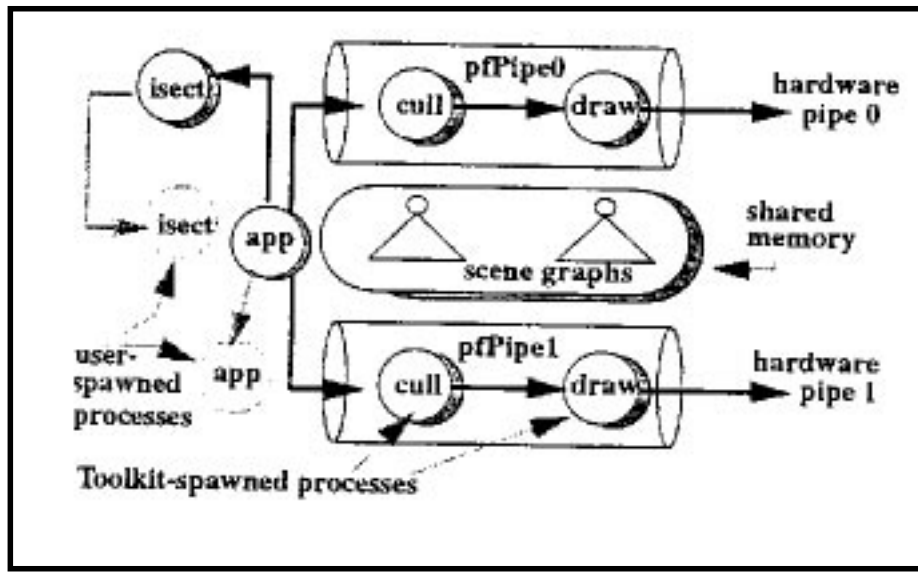
Figure 5: Multiprocessing Configuration. From [ROHL94]

and Z coordinates as well as the direction of view. Direction is given in heading, pitch and roll where heading is rotation around the Z axis, pitch is rotation about the X axis and roll is rotation about the Y axis. All directions used in this document refer to the Performer coordinate system.

2. Networking

The ship virtual environment trainer is designed to be a networked simulation in order to allow many personnel to participate in the training. In order to have more than one work station participate in the ship virtual environment trainer, the different workstations must possess the software code and visual database generated by this thesis. There must also be a communications network between the workstations involved, as well as a standard protocol to send and receive information. The standard protocol used in this application is the standard distributed interactive simulation (DIS) protocol [IST93], which is designed by the government to ensure compatibility between workstations involved in networked interactive simulations.
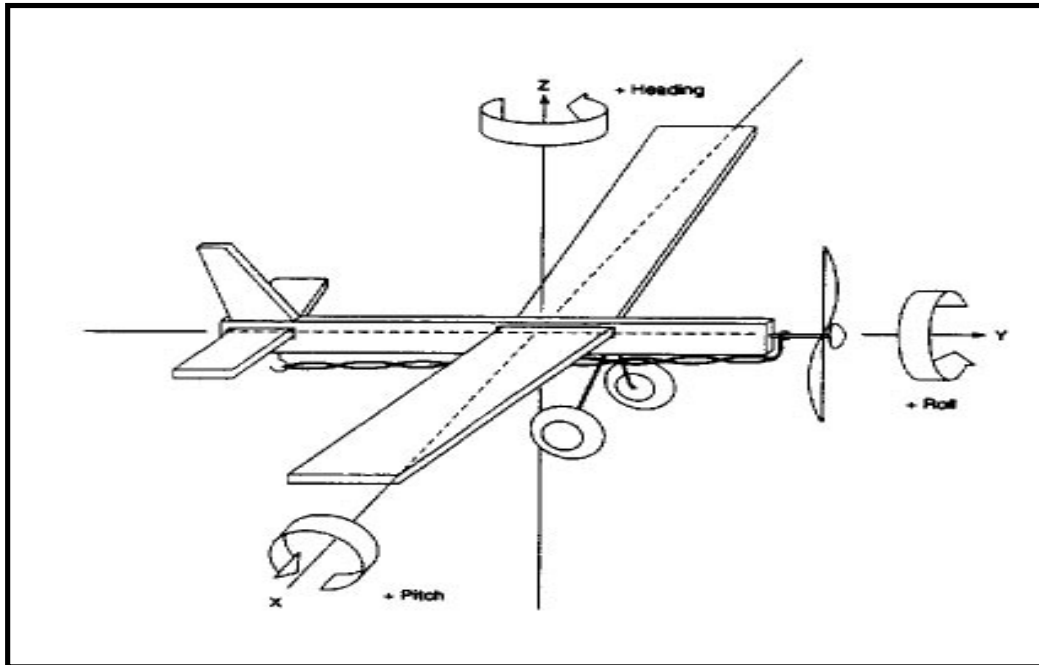
Figure 6: Performer Coordinate System
From [SGI94], used with permission.

The basic concept which this protocol uses is decentralization of the database, so that each individual participant keeps his own copy of the database. Thus, the data which needs to be passed across the network is minimized. Changes to the visual database and locations of participants in the virtual environment are updated periodically by sending protocol data units (PDU's) across the network.

D. SUMMARY

The shipboard VET was designed using many different facets of computer science: modeling the environment, creating code to allow the user to interact with it, and networking the application between workstations. Throughout it all, the overriding design criteria was creating the best possible trainer for use in the fleet. From a computer science standpoint, this normally became what can be done to increase the speed and realism of the application.

III. MODELING

The initial step in creating a virtual environment is to create a model which adequately represents the actual environment and which can be rendered in an acceptable fashion. In order to produce a highly effective training experience for the user, the simulation must be able to make him believe he is actually where the model tells him he is, or at least make him suspend his disbelief. Since the user may be highly familiar with the actual environment represented in the virtual world, the fidelity of the model to the actual ship is vital. There are many methods to create a model with the required fidelity for a shipboard virtual environment, including designing the actual ship using a Computer Aided Design (CAD) tool which was designed to create visualization data as a by-product of the design, importing CAD data into a visualization format, and creating a model based upon ship's drawings using a modeling tool.

The model used in this thesis is the *Antares*, a proposed roll-on/roll-off vessel on which NAVSEA is currently performing feasibility studies. The original model consisted of only the hull and vehicle decks. It was given to the authors by Advanced Marine Enterprises, a Crystal City contractor performing vehicle access/egress studies for NAVSEA. This portion of the model contained approximately 2,000 polygons. The authors added several interior spaces to the model, bringing the final polygon count to approximately 23,000.

A. PREVIOUS WORK

1. Early Work

In-depth studies of visualization databases began with Clark's landmark paper [CLAR76]. In it, Clark, who later founded Silicon Graphics, discusses the advantages of using hierarchical data structures to represent visual databases. Today, most graphical databases are arranged in the format he proposed almost twenty years ago. Rubin and Whitted of Bell Labs applied Clark's theory and created a system to display a hierarchical database visually [RUBI80].

2.   Recent Work

Most of the recent work in computer modeling for walkthroughs has been performed by two groups. Airey, Brooks and others at the University of North Carolina, Chapel Hill, have led the way in precomputing visibility volumes, while Funkhouser, Sequin and Teller at the University of California, Berkeley, have also done important work in this field. Most of the work of both these groups has centered on using potentially visible sets and is discussed in detail in Chapter VII. [BROO86] [AIRE90A] [FUNK94]

Other researchers at UNC have been researching a system very similar to this thesis. Mine and Weber are working on a system to render databases with extremely high polygon counts, specifically portions of submarines, which is described in [MINE94]. This work is less theoretical and more applied than most of the other works on architectural walkthroughs, and describes the design considerations required to build such an application. Their project used many of the same solutions as this thesis, including level of detail generation, potentially visible sets, and instancing.

Another group who created a similar walkthrough is a group of researchers from IBM's T. J. Watson Research Center in New York and the Research Triangle Institute in North Carolina. They created a virtual environment of the pre-World War II interior of the Frauenkirche, a church in Dresden, Germany which was destroyed by Allied air raids in 1945. Their efforts and results are described in detail in [JALI94]. They created a system which displays a stereo representation of a 165,000 polygon, textured model of church using an HMD and large screen displays. Their simulation achieved a frame rate of three to five frames per second in the most complex areas and six to nine frames per second in areas with reduced complexity. The most interesting portion of their work is that they used a SGI Onyx RE2 and the Performer application programming interface, the same equipment and software used in this project.

An additional work which deserves mentioning is [GVU95], which describes a new type of level of detail management. They take into account the fact that the human eye focuses better on objects in the center of its vision than those in the periphery. Exploiting

this fact, they render the objects in the middle of the field of view with higher resolution than those near the edge of the viewing frustum. Although this reference was discovered too late for the theory to be incorporated into this work, this method holds much promise, especially in head mounted displays.

B.   CREATING A DATABASE FROM CAD DATA

When computer systems were fairly slow, the number of polygons which could be included in a real time system was correspondingly small. The disadvantage of this is that the simulations were not very realistic, but the advantage was that it was relatively simple to build such a visual database from scratch. As the speed of systems has increased, the number of polygons which could be rendered has also increased, to the point where it is no longer feasible to use a modeling tool to build large-scale visual databases without originally being designed on a CAD tool. Luckily, the increase in computer power has caused an increasingly large percentage of engineering design to be done using CAD tools. The optimum way to create visual databases is to convert CAD data into a format which can be visualized.

This is the approach that most creators of large-scale virtual environments use. However, CAD data is not in a format which can be easily visualized. One problem is that most CAD tools store the data as a series of lines, vertices, and arcs rather than as polygons, as required by the majority of today's high speed rendering tools. In addition, most CAD databases contain much information that is required by the mechanical engineer but is useless to the computer scientist, such as objects' weights, moments, centers of inertia, etc.

Programs have been written, however, which convert the lines, vertices, and arcs to polygons and strip away extraneous data to produce a database which can be rendered in real time. These are used by most of researchers creating large scale databases [AIRE90A] [FUNK94] [MINE94]. However, all these efforts have converted CAD data which was originally in AutoCAD's DXF format, the most widely used form of CAD data. All CAD formats are different, so converters written for one format will not work for another. Since

writing a converter to transform a form of CAD data to visualization data is a extremely time consuming and difficult task, it is very difficult to use CAD data which is not in a common form.

Originally, the authors intended to create a virtual environment of the DDG-51 class destroyer, the Navy's newest class of combatant, using its CAD database. This task, however, proved impossible for a variety of reasons. The first is that getting the CAD data from NAVSEA proved to be a difficult task, which took approximately nine months to accomplish. The other problem is that each of the primary contractors for the DDG-51 have use their own CAD tool to design and update the ship. This meant that, in order to use the CAD data which was finally supplied, it is necessary to write a converter to put that data into a form which can be visualized. The combination of these problems proved insurmountable, and the thesis was changed to building a visual database of a ship from scratch and designing a way to let the user interact with that ship as realistically as possible.

C. MULTIGEN

The model for this project was created using MultiGen, a modeling tool created by Software Systems of San Jose, California. MultiGen is one of the most capable off the shelf modeling tool currently on the market, and it allows the designers to do many things in the modeling stage which would otherwise have to be done at run time. The model was begun using version 14 of MultiGen [SSI94A] and was completed using version 14.1 [SSI94B], which incorporated additional features which appear in the final model, such as switch nodes. The models created by MultiGen are saved as in the FLT format, which are directly readable by a loader supplied with SGI's Performer.

MultiGen's main advantage is the complexity of the model that designers can build with it. Many of the ideas presented later in this chapter are implemented using MultiGen, and how MultiGen aids in implementing these features is covered in those sections.

MultiGen has a few disadvantages. When it is handling very large databases, it is very slow, and version 14.1 appears slower than version 14. Another disadvantage is its very

high price; MultiGen costs over $50,000 for a single license with the required features, and $25,000 for additional copies, even with an educational discount. This precludes getting additional copies of the program, and results in the several projects which use MultiGen competing for the single license. However, the most glaring disadvantage of MultiGen is its poor user's interface. Although once a designer determines how to use the interface, most operations can be done fairly quickly and easily, there is an extremely steep learning curve. Compounding this, the help system is cryptic and the manuals are too vague to aid the designer. Also, one of the most powerful features of MultiGen was the ability to encode non-geometric data into the database, and how to retrieve this data at run time is not covered in any of documentation. To be fair to Software Systems, help was quickly available through the SGI's Performer list server, and Marcus Barnes of Software Systems sent example programs and databases which demonstrated some of the harder to understand features. However, without this help, much of the power of MultiGen would not have been available to the authors.

D. CREATING THE MODEL TO MAXIMIZE PERFORMANCE

In every stage of building the model of the *Antares*, the primary concern was maximizing the speed of the application. Several methods were used to increase the efficiency of the database in order to increase the frame rate of the application.

1. Hierarchical Data Structure

The visualization database for the ship uses a hierarchical data structure consistent with Silicon Graphics' Iris Performer application development environment. This hierarchical data structure uses a directed acyclic graph (DAG) to store the visual database, often referred to as the scene. In order to understand the advantages of such a system, it is first necessary to reiterate how Performer determines which portions of the scene is rendered in each frame.

Because of the nature of Performer's cull process, which is described in detail in Chapter II, organizing a database's DAG based upon physical locality rather than a logical

grouping is much more efficient. For the example of the town database shown in Figure 7, a logical grouping of the database would put all buildings in the one branch of the DAG and all trees in another, as displayed in Figure 8. However, a spatial organization, as shown in Figure 9, is much more efficiently culled. If the user's location is in tile 5 and he is looking towards tile 9, and the logically arranged database in Figure 8 is being used, none of the nodes can be culled at the first level, since there is a tree and building in the viewing frustum. This means that the cull process has to traverse to the lowest level in the branch under "trees" and cull each tree separately. If the same situation were to occur and the spatially arranged database of Figure 9 is being used, then immediately tiles 1-4 and 6 are culled at the first level below the root node, which greatly reduces the cull time, and therefore the rendering time.
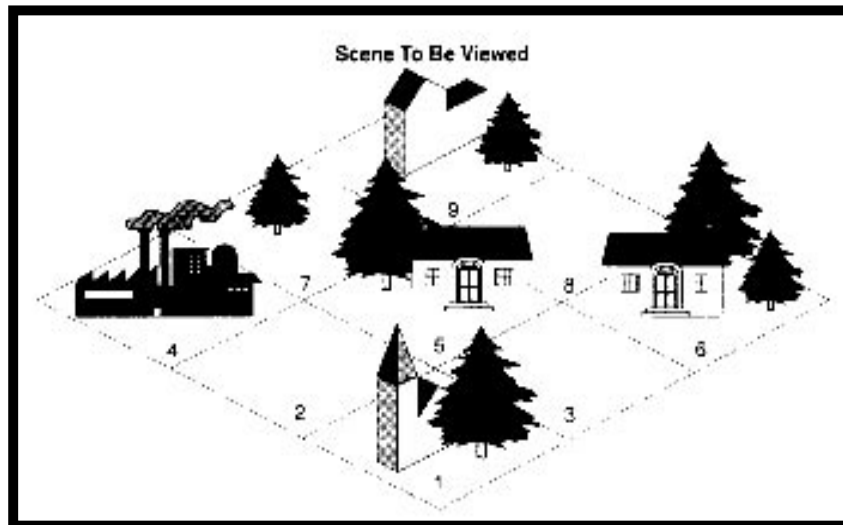
Figur e 7: Typical Town Database
From [SGI94], used with permission.

The database for this project was divided spatially as much as possible. Each compartment was placed into its own group, so that the entire compartment and all that it contained will be culled quickly if it should not be rendered. The only exceptions to this philosophy were the vehicle decks and the exterior portions of the ship.
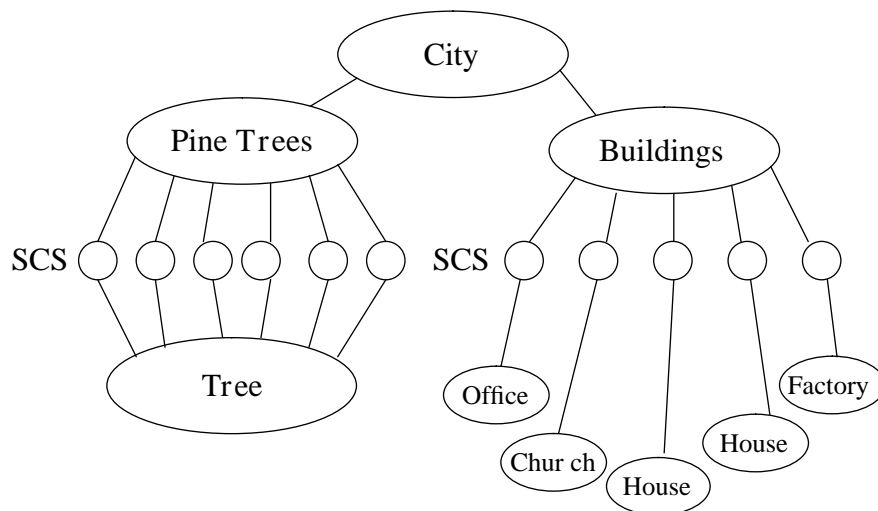
Figure 8: Inefficient Tree Structure
From [SGI94], used with permission.
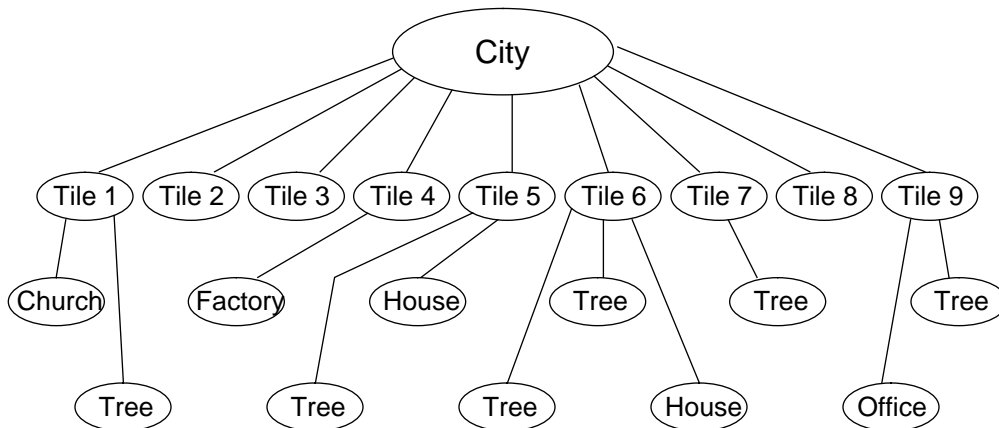
Figure 9: Efficient, Spatially Partitioned Database Organization
From [SGI94], used with permission.

MultiGen can display a hierarchical representation of the model, which facilitates creating the model in the form of a DAG. Also, MultiGen allows the designer to maneuver objects easily from one portion of the DAG to another. This lets the designer build the model in a logical manner, keeping like objects in the same portion of database for ease of use while creating the database, and then move the objects so that the database is spatially ordered to increase run time performance.

2. Modeling Performance Enhancements

   a. Levels of Detail

   The amount of detail of an object that a human can make out depends upon many factors, including the size of the object, its distance from the viewer, its position in the viewer's field of view, among others. Levels of detail (LOD) exploit this phenomenon to increase the speed and realism of a virtual environment.

   If the user is only a short distance from an object, he can make out every detail of the object, so that object needs to be rendered as minutely as possible. However, if the user is a great distance from an object, he can only make out gross details of the object, perhaps just a vague outline of its shape. If the same detailed model of the object is used when the user is this distance from an object, it still contains the same number of polygons, even though most of them cannot be made out by the user. This means that the application must perform the same amount of time processing it, even though that additional time adds little or nothing to the realism of the scene.

   LOD's exploit the fact that the further an object is from the user, the less detailed it needs to be, by displaying different representations of the same item depending upon the object's distance from the user. If the user is close to an object, it is rendered using a highly detailed, high polygonal count representation to give the user a realistic looking object. If the object is a great distance from the user, a low detail, low polygonal count representation is rendered, since the user could not make out any more detail, even from a detailed model.

   Performer handles LOD's during the cull process. When the process encounters an LOD during its traversal of the database, it first determines if this is the correct representation of an object to display, based upon the user's distance from it. If this LOD is displayed, Performer then treats it as any other node, and determines whether it to cull it from the database. If the LOD is not in the correct range to be displayed, it is immediately culled.

Levels of detail are used as much as possible in the model Antares. Round objects presented the best opportunity to reduce polygonal count by using LOD's, since it is easy to create several representations of the same object using cylinders with different numbers of sides. In most cases, four different representations of each round object are used, the highest LOD consisting of a twenty-four sided cylinder, with the others, in decreasing order of detail, consist of a twelve sided cylinder, a six sided cylinder, and finally, a cube. To determine how many sides should be used, different sized round objects were created with different numbers of sides. These were visualized from various distances to determine not only how many sides to use, but also what distance to switch from one level of detail to another when approaching or moving away from an object. This is important because the transition must be seamless, that is, unnoticed by the user. Figures 10 - 14 shows example LOD's from the ship database.
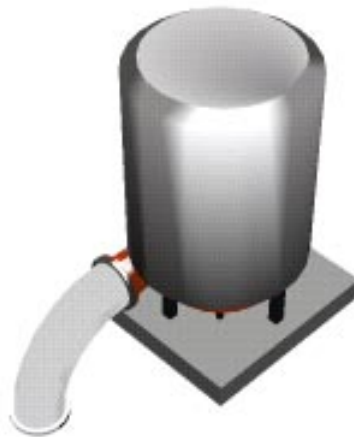
Figur e 10: Main Feed Pump, Maximum LOD
427 Polygons

MultiGen is designed to make using LOD's easy. The modeler simply creates several Level of Detail nodes, each of which contains the geometry for a different
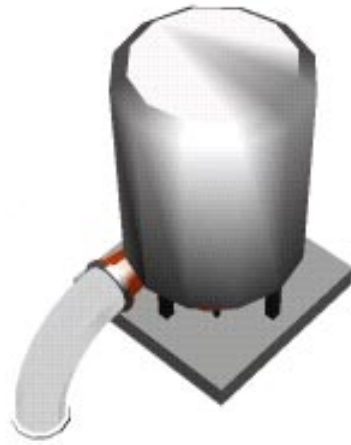
Figur e 11: Main Feed Pump, Med-Hi LOD
248 Polygons
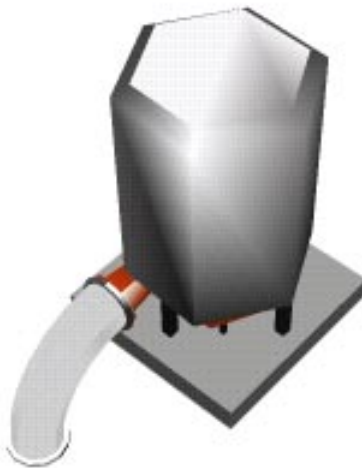
Figur e 12: Main Feed Pump, Med-Low LOD
129 Polygons
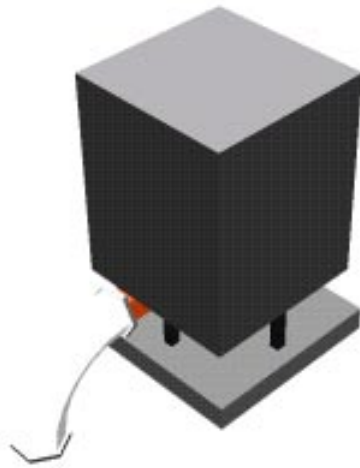
Figur e 13: Main Feed Pump, Minimum LOD
43 Polygons

Eye Point

Object Not Rendered

| High LoD | Med-High LoD | Medium-Low LoD | Low LoD |

Far Clipping Plane

Main Feed Pumps

| High LoD | Med-High LoD | Medium-Low LoD | Low LoD |

Main Feed Booster Pumps

| High LoD | Medium LoD | Low LoD |

Deaerating Feed Tank

0   5   10   20   50
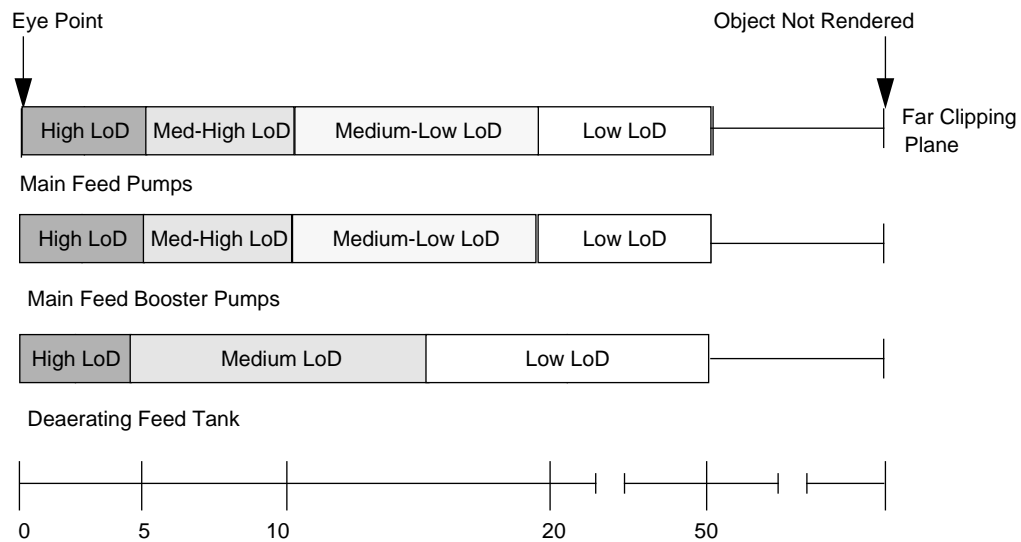
Figur e 14: Level of Detail Ranges (meters)

level of detail. The modeler enters the minimum and maximum distances he wants each
LOD to be visible in the LOD node, which contains space for this data. In order to calculate

31

the object's distance from the user's viewpoint, the object must be given a single point as its location. MultiGen automatically calculates the center of the object to use as this location, or the modeler can enter his own value if the center of the object is not the optimal solution. During run time, the distance is measured form the viewpoint of the user to this location to determine which of the LOD's to display.

The effectiveness of LOD's was found to be reduced in a shipboard environment compared to most other types of visual databases. Ships are extensively subdivided to give watertight integrity and limit battle damage. In addition, most areas are extremely crowded with equipment. These two factors result in the interior of a ship having very short sightlines, which reduces the impact using LOD's has on performance. In many cases, if the user can see an object at all, he is close enough to require that the highest level of detail be rendered. Also, by using potentially visible sets (PVS), objects far away are not even included in the tree which is traversed by the cull and draw processes, so the increase in performance using LOD's gives is reduced. However, they still increase performance in PVS with large volumes, such as the engine room, even if they are not visible, since the draw process has fewer polygons to remove from the scene.

        b.   *Instancing*

Instancing is a method to reduce the amount of data required to be stored in memory while running the application. In many models, the same type of item is present more than once in a scene. For example, a classroom scene will have approximately thirty student desks, each almost exactly the same as the others. Instancing takes advantage of this phenomenon to store the physical geometry of the desk into memory only once and copy that one desk twenty-nine times to create all thirty desks in the classroom. In this way, instancing greatly reduces the amount of memory required to display the scene, and reduces the number of times that the application will have to swap items from memory to disk, thus enhancing the simulation's frame rate.

Instancing's disadvantage is that the polygons of an instanced object are stored in a local coordinate system, and transformations must be applied to determine where in the world coordinate system that copy of the object is. In the above example, every polygon in a desk would have its position in the local coordinate system multiplied by the transformation necessary to move it from the location of the single desk stored in memory to its location in the world coordinate system. These additional transformations are added to both the cull and draw processes, increasing the time of those processes and lowering frame rate. This is not a disadvantage if the object is dynamic, that is, it has the freedom to move in the virtual world, since these objects must be transformed every frame because of their freedom of motion. Thus, the decision to use instancing on static objects must be based what factor is limiting frame rate. If an application has such a large number of the same object in a scene that is limited by memory swaps, then it is worthwhile to instance static objects. However, if the application is not so limited, instancing static objects merely adds additional overhead, slowing the application, and thus should be avoided.

Based upon these considerations, instancing is used only for doors in the **Antares** model, since they are the only dynamic objects which appear several times in the model. Since the doors have only five polygons each, the savings in memory is minimal, but instancing was still done as a proof of concept which could be easily applied to a model where the savings might be more substantial.

c.   Textures

Another method used to increase the realism of the environment was texturing. Textures are images stored in a variety of formats which are attached to polygons in the models. These polygons then display the image when the user views them in the virtual world. This is an extremely effective technique in creating a highly realistic virtual environment, because it produces a photorealistic effect using far fewer polygons than are required to create a less realistic effect. Figure 15 shows a control panel for a console in CIC which is created with a texture taken from a photograph of the actual panel, while

Figure 16 shows a control panel created using polygons. The textured control panel looks much more realistic than the control panel created with polygons, yet the textured version is made up of only one polygon, while the other consists of close to one hundred polygons.
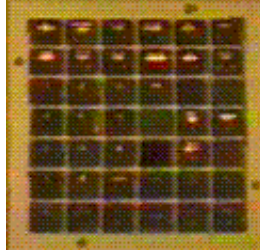
Figur e 15: Contr ol Panel Cr eated with T extur e
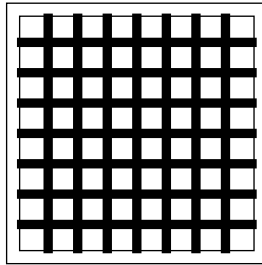
Figur e 16: Contr ol Panel Cr eated with Polygons

Texturing has three major disadvantages. The first is that most textured images lose their clarity if the user approaches them too closely. This leads to a a decrease in the user's feeling of being immersed in an environment, since the realism is reduced under fine examination. For pattern textures, this problem can be overcome by using detail textures, which combine two textures during close examination to maintain the texture's appearance. For textures which represent actual objects, such as Figure 15, the only method to prevent this is to use textures which are scanned with a very high number of dots per inch (DPI). However, the memory requirements for this type of textures are very high, and using several of these textures will quickly use all available texture memory.

The second disadvantage of texturing is that apparent 3-D images are really flat and this makes it is impossible for the user to interact with objects on the texture. For example, if a texture is used to represent a beam on a bulkhead of the ship, the user can see

34

the beam as it appears to come out of the bulkhead in three dimensions. However, if the user is walking along the bulkhead, he cannot run into the beam as he would in reality. In the same vein, the user cannot interact with textured control panel in Figure 15, while he could enter commands using the polygonal control panel in Figure 16.

The third disadvantage is that the lower-end machines do not have specific memory or hardware to handle textures as do the more expensive Reality Engines and Reality Engine 2's. Although this is not noticed on the high level machines, for which this application is designed, a large degree of realism is lost when it is run on the lower end machines, such as the Silicon Graphics Indigo 2. These machines do not apply textures to polygons which are textured in the database.

Despite these disadvantages, textures add greatly to the realism of the environment and were used extensively in this thesis. The majority of these images are photographs of objects which would normally be found on a ship. These images were obtained from several sources, including the Naval Postgraduate School texture library, other researchers working on similar projects who were willing to share their textures, and the authors' photographs taken on a visit to the USS *Enterprise* (CVN-65). These photographs were scanned into the Computer Science Department's Macintosh IIci computer using Adobe Photoshop and then transferred to one of the Graphics Lab's Indigo 2 workstations. There they were converted into RGB files, which were added to objects in the virtual world using the MultiGen modeling tool.

  d. Dynamic Objects

There are several objects in the database which are designed to move when the user interacts with them. These include doors, valves, and vari-nozzles, among others. Performer refers to these objects as Dynamic Coordinate System (DCS) nodes, while in MultiGen they are called Degree of Freedom (DOF) nodes. The difference is minimal, since when the loader loads a MultiGen FLT file into Performer, it converts DOF objects into DCS nodes.

DOF nodes are easy to build using MultiGen. The DOF node is created by selecting an icon and the modeler then adds the geometry of the dynamic object as its child. The DOF node is then positioned in the model, which involves selecting the point which all translations and rotations are based upon. On a lower level, this is the point which is translated to the origin to perform all rotations, and the point which, when the object is translated to point (x, y, z), is actually at (x, y, z) in the world coordinate system. After positioning the DOF node, the modeler then limits the amount the object can rotate and translate in each of the three dimensions. For example, doors in the database can be set to rotate only 90° and are not allowed to translate at all, matching the characteristics of physical doors. How this information is accessed will be discussed below.

e. *Switching Objects*

In an active simulation, there are times when the database is updated based upon events which occur in the virtual world. These include removing an object, adding an object, or replacing an object with another. For example, this may be necessary for terrain databases due to shifts in the topography due to earthquakes, mudslides, fires, weapons detonations, etc. If the change in the model is great enough, it is easier to just delete the original model and replace it with another which reflects the new conditions. However, this will take a great deal of time, and causes a noticeable lag in the simulation. Therefore, unless the alteration to the virtual world is extreme, this method is avoided. A better method of changing the database is to predict all the changes which might occur, and create alternate objects which reflect these changes. Then, the objects in the original database which need to be changed due to the user's actions are replaced with objects which show the required changes. For example, in a driving simulation where collisions are possible, the original car model is free of defects. If the user's car is hit by another, the fender of the original car is removed and a fender with a dent is put in its place. The transformation happens so fast that the user is unaware of it; to him, it appears that the original fender has been dented.

The newest release of MultiGen, version 14.1, includes features which allow this type of change to be implemented quite easily. It allows the modeler to create a switch node, under which he puts both all the geometry which will appear in that area. He then creates different masks which contain all or part of the geometry under the switch node. In the above car example, the mask for the damaged fender contains the polygons which represent the dent, while the mask for the undamaged fender contains the polygons which represent a smooth fender. Both, however, contain the same polygons for the headlight, since the damage does not affect it.

The **Antares** model uses this type of switch node to change the virtual environment after various casualties occur. A small piece of the bulkhead behind the area where the fire occurs has been removed and replaced with a switch node. (Currently, the fire only occurs at one location near the after bulkhead of the lower level of the engine room.) Originally, the polygon in Figure 17 is displayed. This polygon has the same texture as the bulkhead, so that the area appears to be a contiguous part of the bulkhead. As the fire starts, that polygon is replaced with the polygon in Figure 18, which represents slight damage. If the size of the fire surpasses a specified limit, that polygon is replaced with the one in Figure 19, and if the fire surpasses a higher limit, that polygon is replaced with the polygon in Figure 20. Thus, the damage caused by the fire is based upon the extent of the fire and adds to the realism of the virtual environment.
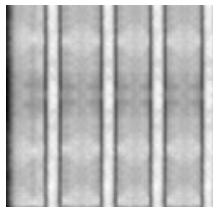
Figure 17: Undamaged Bulkhead

E. LOADING CALLBACK

One of the most powerful features of MultiGen is the ability to embed non-visual information into the visual database. This greatly increases the potential functionality of the application for which the model is created. This feature is used extensively in the shipboard

Figure 18: Minimally Damaged Bulkhead

Figure 19: Moderately Damaged Bulkhead
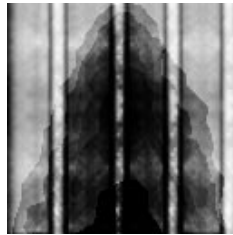
Figure 20: Severely Damaged Bulkhead

VET. It is the key to the hypertext capability, potentially visible sets, collision detection, and the manipulation of objects. Without this capability, all of these features would have been severely degraded, if not impossible to implement. The key which unlocks this power is the loading callback, but before that can be discussed, it is first necessary to describe how MultiGen stores non-geometric data in the visual database.

1.   MultiGen's Embedded Data Structure

MultiGen has a variety of nodes, some which are very specific, others which are fairly generic. The former include the degree of freedom (DOF) node, the level of detail (LOD) node, and the switch node. These nodes normally only contain the information necessary to handle their special functions. However, the generic nodes, which are the

group node and the object node, contain data field into which the modeler can embed data. These include two integer fields, named special fields, which the modeler can enter integer values into, and a string field, where the modeler can enter text data. Although the space to store data is fairly limited, the modeler can convey quite a bit of information, if he does it correctly.

2.   Reading the Data at Execution Time

The data stored in the model is read by the loading callback, a user-defined function which accesses the information stored in the database. How this function works is very simple in theory. The initialization procedure sends the callback to the loader prior to loading any models. The FLT loader reads the hierarchical data structure of the model in a depth-first traversal, and the callback function is executed on each node as it is loaded.

Based upon what type of node is being loaded, the callback takes different actions. If it is a geometry node, no action is taken. If the node is a DOF node, the callback extracts the rotation and translation information and a pointer to the node and stores them in a matrix. If the node is a group node or an object node, the first special field is read to determine what the group or object contains. There are six possibilities:

- PVS Node: The node is the root for all the geometry in a PVS. In this case, the callback stores a pointer to this node in a matrix of such nodes. The second special field contains index where the node must be stored.

- Manipulate Node: The node contains an object which can be manipulated by the user. In this case, a pointer to the node is stored in a matrix of such nodes, under the index specified in the second special field. Also, a pointer to the node and the intersection mask value of a manipulate node are stored in another matrix, named "treeMask," whose function will be described later.

- Information Node: The node contains its name and function so it can be displayed in the hypertext window. In this case, a pointer to the node is stored in a matrix of all the information nodes, under whatever index is next. Also, the name and function of the object are read from the text field of the node and stored in this matrix. Additionally, the pointer to this node and the intersection mask value of an information node are stored in "treeMask."

- Ladder or Deck Node: This node contains the name of the function of the deck or ladder. The data and pointers are stored just as an information node, but the intersection mask sent to "treeMask" is different. This is because the user can walk

39

on ladders and decks, but not the other parts of the database.

- Door Node: This is the parent node of a DCS node which contains a door. A pointer to the node and the index of its door in the DCS matrix are stored in a special matrix containing all the door information.

- Valve Node: This is the parent node of the two DCS nodes that each valve contains. A pointer to the node and the indices of its valve stem and handwheel in the DCS matrix are stored in a special matrix containing all the valve information.

After the entire model has been loaded, the treeMask matrix is used. Each node in treeMask is assigned its associated intersection mask. This ensures that every part of the database will have the correct mask, so that the collision and picking mechanisms will work correctly.

Without the loading callback, it would be much more difficult to interact with the database. The only way to get information about objects in the model would be using the names of objects, which is a very slow, tedious process of limited scope. This capability greatly increases the functionality and versatility of the simulation.

F. SUMMARY

The modeling portion of this thesis explored several methods to maximize performance by decisions made during the modeling process. It pointed out that increasing performance is not solely the responsibility of the software writers, but begins with a well designed and constructed model. Performance, realism and accuracy must be the foremost considerations in every step of the modeling process.

To maximize performance in the *Antares* model, levels of detail, instancing and textures were used. While levels of detail and instancing added little to the performance of this application, textures greatly added to its realism while reducing polygonal count to enhance performance. Also, much of the functionality of the application is embedded in the model. This includes dynamic objects, switch nodes, and all the information for potentially visible sets, hypertext displays, objects which can be manipulated, as well as all the collision masks.

IV. INTERFACE CONTROLS AND DISPLAY

This chapter summarizes the presentation medium and interface controls with the ship virtual environment. Methods to interface with and display virtual environments change rapidly as technology improves. There is no definitive "best method" to display and interface with a virtual environment. Operations research and psychology personnel at places like the University of Washington's Human Interface Technology Laboratory (HITL) iteratively analyze various virtual interfaces. They attempt to create general rules which virtual worlds designers can then apply to their applications, and then analyze the results to determine the effectiveness of the results.

A. IDEAL INTERFACE

The ultimate interface goal is to provide the user with a sense of "presence", that he is actually inhabiting a new place instead of looking at a picture. The user should be able to naturally navigate through the virtual world the same way he maneuvers through the natural world. The user should be able to open doors, walk down stairs, look in any direction, just as he can in reality. Certainly, it would be more realistic to actually walk through the virtual world rather than sit at a console and use an input device to maneuver with. Likewise, it would be more realistic to actually feel the virtual object that the user grabs. The ultimate interface would be one which the user cannot differentiate from reality, a sort of "Is it real, or is it Memorex?" condition. Although far from achieving this ultimate interface, substantial strides have been made toward it.

The interface between the human and the machine should be configured to match the sensory and perceptual capabilities of the human as well as provide a sense of "presence". Currently, one of the best interfaces which provides these characteristics is the head-mounted display (HMD) system shown in Figure 21. A personal 3-D "cinerama theatre" is created within the headgear. A tracking system attached to the headgear tracks the viewer's position and view direction to display the corresponding view in the virtual world. A data

glove provides a means for the user to interact with objects in the environment through gestures. A voice recognition input device also provides the ability to interact with the system. An acoustic virtual display provides sound that is spatially oriented. A tactile response can also be provided through the use of vibration transducers in contact with the skin of the hand or body. Tactors may be actuated as a function of the shape and surface features of the virtual object and the instantaneous position of the hands and fingers. [HITL94] While not all HMD's are this advanced, all contain the means to display an image which takes up most of a user's field of view and translate the user's head motions to the computer coordinating the displaying.
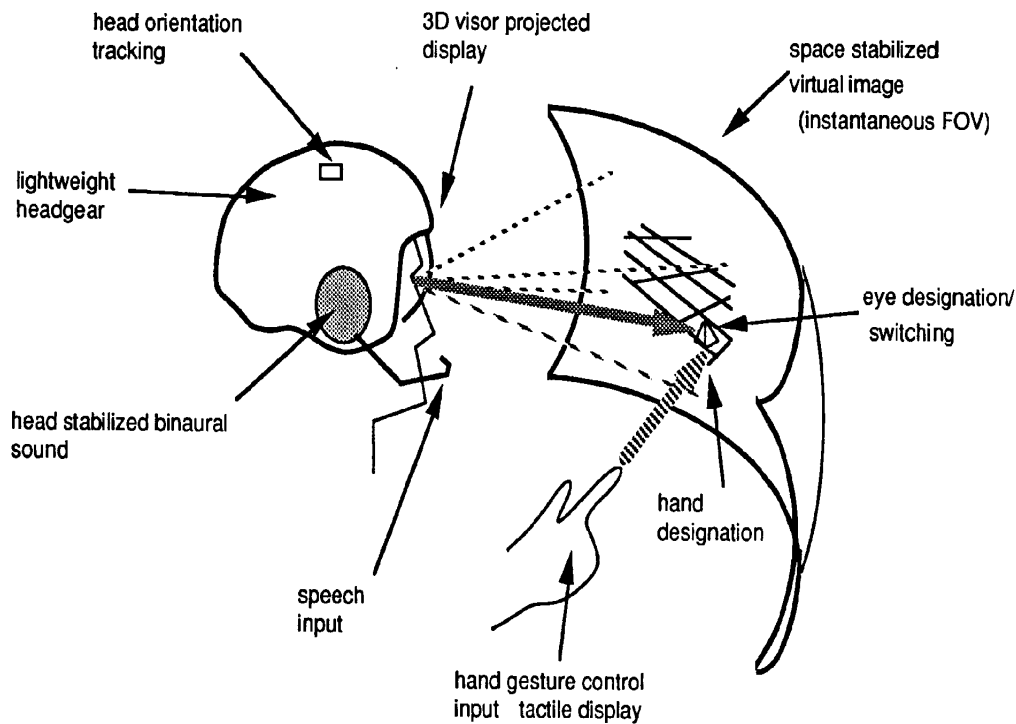
Figure 21: Head-mounted Display System. Fr om [HITL94].

42

However, although many in the field of virtual reality have indicated that interfaces such as this will be commonplace in the near future, this type of ideal interface exists only in a few experimental laboratories. Although HMD's have been created which operate well and reliably, all the other equipment is far from being perfected. Datagloves to transmit hand gestures to the computer have been mildly successful, but they are notoriously unreliable, and are several years from becoming commonplace. Tactile feedback devices are still in their infancy, and have had very little success so far. Most voice recognition systems have to be tailored to a single user, and even then are only mildly successful after much practice. In short, this ideal interface exists only in optimistic visions of the future. An application built today to be used in the field, not a laboratory, will have to use interfaces and displays which fall far short of this ideal.

B. DISPLAYS USED IN THE SHIPBOARD VET

Presently, there are two formats in which the shipboard VET is displayed to the user. The application's output can be directed to a traditional monitor or an HMD. While the HMD gives a much better image of being immersed in the environment, many people cannot currently wear HMD's for long periods of time. In addition, current low-cost (under $25,000) HMD's do not have the resolution necessary which allows the user to read text. This limitation would make the hypertext display impossible, thereby severely limiting the functionality of the application. Also, Levit and Bryson point out that many people, for a variety of reasons, will at least occasionally want to use a desktop version of the application [LEVI92]. Therefore, versions of the shipboard VET were created for both types of displays.

The standard output display is a full screen display which includes the virtual scene display, graphical user interface (GUI) and deck overview. A pop-up window, which displays information about objects in the virtual ship, is displayed when objects are selected with the mouse. These displays and there relative locations on the screen are shown in Figure 22 and Figure 23.
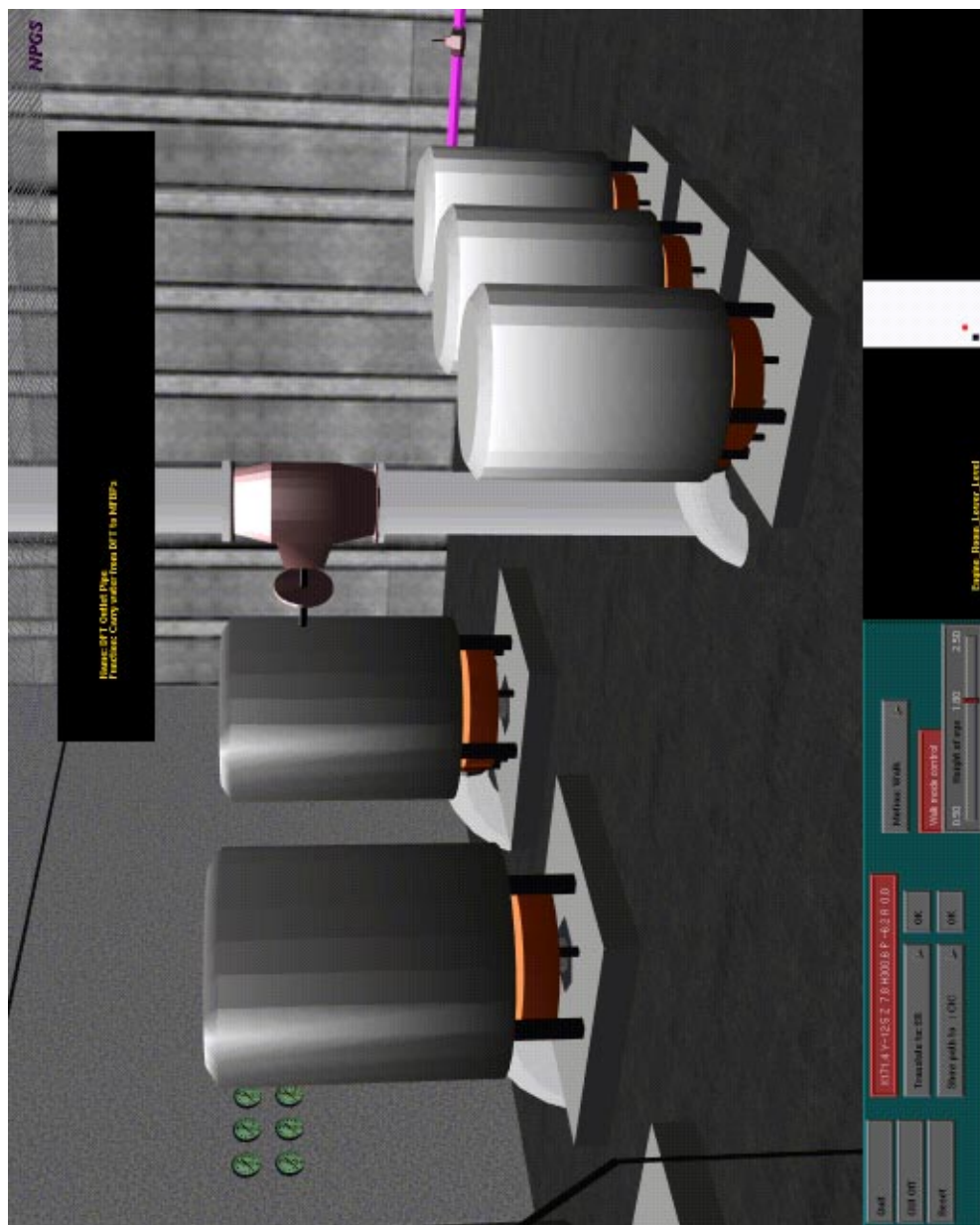
Figure 22: Screen Display, Antares Engine Room
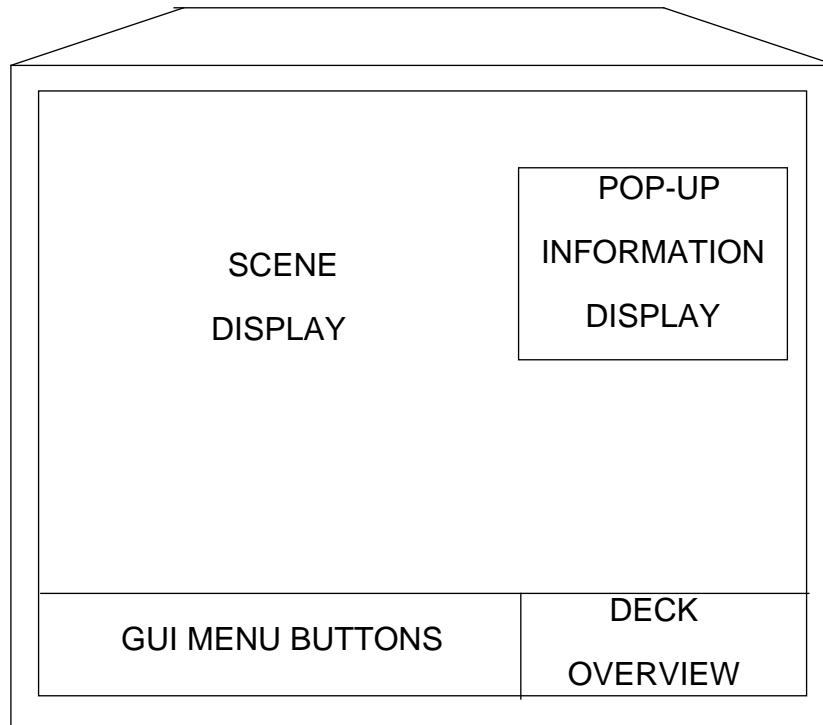
Figur e 23: Monitor Display

1.   Scene Display

The scene display consists of the whatever objects in the visual database that fall within the viewer's horizontal and vertical field of view, and between the far and near clipping plane. Objects not within these bounds are not rendered. This volume is called the viewing frustum and was described in more detail in Chapter II.

When displaying the scene on a monitor in the standard output mode, the horizontal field of view is set to forty-five degrees and the aspect ratio to 0.8. The vertical field of view is calculated by the application based upon these values and window size, and for standard monitors this value is thirty-six degrees. Forty-five degrees was chosen as the horizontal field of view since this is the normal focusing field of view for humans. The near clipping plane is set to 0.01 meters and the far clipping plane is set to twice the scene size to ensure nothing is clipped from the scene display. Objects outside this range cannot be focused on by humans, so they are not displayed. The display space was allotted ninety per-

cent of the monitor area for two reasons. The first is that the larger the virtual scene, the greater the user's sense of immersion. Also, most of the information the user requires is conveyed in this area. This leaves the graphical user interface and deck overview areas with ten per-cent of the screen, adequate for the application's needs.

When using a head-mounted display to view the virtual ship, the field of view is changed to take advantage of the wider display area. The horizontal field of view is set to one hundred degrees and the vertical field of view is set to thirty degrees, which corresponds to an aspect ratio of 0.3. When directing the output to an HMD, the graphical user interface, deck overview and pop-up information window are not displayed. Displaying these would interfere with the already limited viewing area of the HMD and result in a degradation of the feeling of being immersed in the environment.

2.   Deck Overview

Because Navy ships are so large and complex, a deck overview display is provided to aide the trainee in finding his way around the ship. The deck overview channel is located on the lower right hand portion of the screen as shown in Figure 23. It provides an overhead view of the deck on which the user is presently. The deck lay-out is graphically displayed in two dimensions showing the locations of ladders, bulkheads, doorways and passageways. A black position cursor shows the user's position in the virtual ship and moves as the user moves along the deck in the virtual environment. The deck overview display changes to display a new deck when the user changes decks in the virtual environment.

3.   Pop-Up Data Display Window

The maze of pipes, valves, electrical components, wiring and control panels which exist on board a Navy ship is overwhelming to a newly reported sailor. In order for him to obtain a better mental picture what objects are and how systems are inter-related, the user has the ability to select an object with the mouse and display a variety of functional information about the object. When the user selects an object, the pop up window

containing the information is displayed in the upper right hand corner of the screen as shown in Figure 23. To select an object, the user places the mouse cursor on an object and depresses the middle and either the left or right mouse button at the same time. The display stays on the screen until the mouse buttons are released.

4.   Graphical User Interface

The graphical user interface (GUI) provides the user with an "easy to use" menu interface to perform an assortment of functions. It also provides a display of the virtual world coordinates including the user's heading, pitch and roll. The GUI is located on the lower left corner of the screen as shown in Figure 23. A representation of the GUI is displayed in Figure 24. The functionality of the devices on the GUI are described below, starting in the upper right corner and proceeding clockwise:

QUIT		X Y Z H P R		WALK

GUI OFF	TRANSLATE TO<u>CIC</u>	OK

HEIGHT OF EYE

RESET	SHOW PATH TO<u>OPS</u>	OK
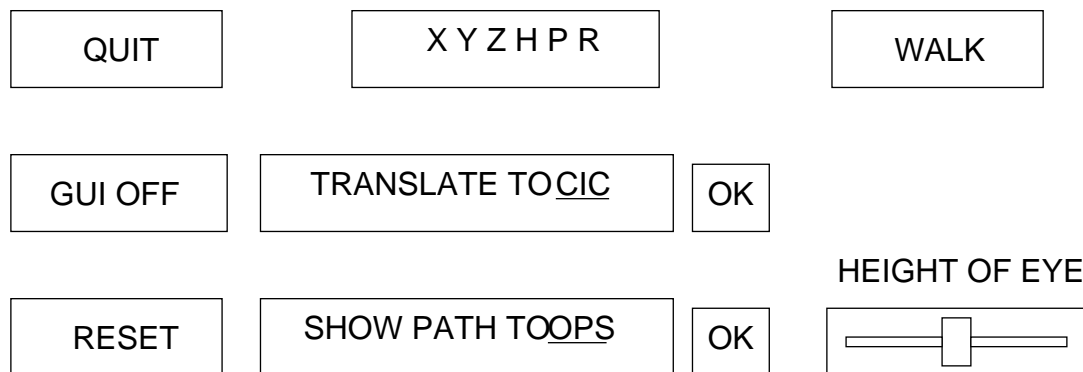
Figur e 24: Graphical User Interface

a.   *Quit Button*

Causes the user to leave the application. Pressing the ESC key also accomplished the same function.

47

b.   User's Position Display

The user's location in three space is displayed here as X, Y, and Z coordinates. Also, the user's heading, pitch and roll are also displayed.

c.   Traversal Mode Selection

There are two methods to navigate through the virtual ship. The default mode is the "walk" mode in which the user moves through the ship at normal walking speed at a selected height of eye above the deck. Collision detection is enabled during walk mode which means that the user cannot walk through objects such as bulkheads and doors. The other traversal mode is 'fly' mode, called this because it is similar to controlling an aircraft. The user can move to any position in the virtual ship in fly mode. Collision detection is turned off during fly mode which allows the user to penetrate objects.

d.   Height of Eye Control

The height of eye control slider is added to enable the user to vary the eye point height above the deck while in walk mode in order that objects which are close to the deck or up high could be viewed at a closer distance. The user can vary height between 0.5 and 2.5 meters.

e.   Path Planning Selection

A path planning tool is provided which takes the user along a path from his present location to a location of his choosing via the optimal route at normal walking speed. The locations which can be selected include CIC, the Radar Room, the Operations' Office, DCC, the Hull Technician's Shop and the ladder to the Engine Room. The user selects his destination by clicking the button titled "Show path to: <destination>." Each time he clicks it, a different destination is displayed. When the desired destination is displayed, the user selects the "OK" button next to it and begins travelling to that destination. At this point, the menu button changes to "Stop walking to: <destination>", and if the user selects it, he is no longer transiting to the destination and he reagins control of his own motion.

f. Reset Button

This button allows the user to reset the application to its original state. All objects are returned to their initial position, all casualties are terminated, any damage caused by casualties is repaired, and the atmosphere is cleared.

g. Toggle GUI Button

The "GUI-Off" menu button turns the GUI and the deck overview off providing more screen display for the scene. The GUI and deck overview can be returned to the screen display by depressing "F1" on the keyboard.

h. Translation Selection

To facilitate the user the ability to quickly "jump" from one location in the virtual ship to another in the virtual ship, a translate menu button is provided. Preset anchor points to key locations are embedded in the software code. These locations include Combat Information Center (CIC), Damage Control Central (DCC), Engine Room, Bridge and the Vehicle Loading Deck.

C. INPUT DEVICES

The mouse provides many input functions in the virtual ship trainer. Using the screen coordinates of the mouse cursor position and the navigation algorithm discussed below, the user looks around the virtual ship simply by changing the relative mouse cursor position on the screen. The mouse buttons permit the user to change speed and direction of motion. They also provide the capability to interact with objects. When the middle and either the left or right mouse buttons are depressed at the same time, the object pointed to by the mouse cursor is selected. If the object can be manipulated, it is; if not, the data associated with the object is displayed on the data display screen. The mouse also allows the user to select the menu options available on the GUI.

The keyboard is primarily used to initiate casualties on the user in the virtual environment. It also provides another method to accomplish some of the functions which are provided by the GUI. The keyboard inputs and their functions are listed in Table 1.

Keyboard Input	Function
F1	Displays GUI and Deck Overview
ESC	Exits program
'd' or 'D'	Toggles CPU and graphics statistics
'f ' or 'F'	Initiates fire casualty sequence
'p' or 'P'	Places fire nozzle back to stored position
's' or 'S'	Initiates steam leak casualty
't' or 'T'	Toggles texture display
'w' or 'W'	Toggles wireframe display
Shift 'b' or 'B'	Translate to Bridge
Shift 'c' or 'C'	Translate to CIC
Shift 'd' or 'D'	Translate to DCC
Shift 'e' or 'E'	Translate to Engine Room
Shift 'p' or 'P'	Translate to Vehicle Loading Platform
Print Screen	Saves RGB image of display on screen

Table 1: Keyboard Inputs and Functions

D. NAVIGATING THROUGH THE VIRTUAL SHIP

There are two modes to navigate through the virtual ship: fly mode and walk mode. The mode of navigation is limited by what type of virtual display is being used. The walk mode is the only navigational mode available when wearing a head-mounted display. With a monitor display, the user can select either walk or fly modes. Each mode is discussed below.

1. Walk Mode - Monitor Display

Natural walking is simulated in the ship environment with the aid of a mouse. By depressing a mouse key, the user gains speed and translates through the environment. The right mouse button increases forward velocity and the left mouse button increases reverse velocity. Speed can be increased to a maximum speed set to simulate fast-walking by maintaining either of mouse buttons depressed. The middle mouse button causes the viewer to stop and resets speed to zero.

The direction of motion is determined by the view direction, or "heading" of the user in the "X-Y" plane. The view direction is changed by varying the mouse position relative to the center of the field of view on the monitor. The farther to the right of center the mouse cursor is, the quicker the individual will turn to his right.   Likewise, the user can turn to his left with the mouse cursor positioned to the left of center screen There is a one inch box in the middle of the screen referred to as the "dead zone" in which the mouse cursor, if inside this area, does not cause the view direction to change.

The pitch or "up and down" view direction is also determined by the position of the mouse cursor relative to the center of the screen in the vertical direction. With the mouse cursor below center screen, the view direction moves down and likewise, if the mouse cursor is above center screen, the view direction moves up.   The range of motion is capped to straight up (+90 degrees) and straight down (-90 degrees).

The user's height of eye is maintained at a constant height above the deck, which the user selects using the GUI. This allows the user to get different perspectives of the same object.

2. Walk Mode - Head-mounted Display

The method in which the user walks through the virtual ship when wearing a head-mounted display is very similar to the walking method with the monitor display. The only difference lies in the method in which the view direction is determined when wearing a head-mounted display. The head-mounted display tracking device translates the HMD's

51

direction of view to an appropriate view direction in the virtual environment. Therefore, to walk around the virtual ship, the user physically looks in the desired direction and depresses the appropriate mouse buttons.

3.   Fly Mode

The fly mode is similar to controlling an aircraft. In this mode the user can fly through the virtual environment, changing heading, pitch and roll based on the mouse cursor position relative to the center of the screen. The user's speed is determined by mouse button operations in the same manner as it was for the walking methods discussed above. When in fly mode, collision detection is turned off, allowing the user to fly through objects.

E.   SUMMARY

The shipboard VET is designed to be used by sailors at sea. Therefore, its interface does not use most of the equipment included in the ideal interface described at the beginning of this chapter. Most of that equipment has not reached the stage of its development where it has the required reliability to be used outside of a laboratory. Instead, the shipboard VET is designed to be used with displays and interfaces which have demonstrated their reliability in the actual conditions in which they will be used.

The interface was designed to be fairly intuitive, so that minimum instruction is required before the user is able to navigate the virtual environment. The requirement was achieved; most people using the simulation can move around to ship without too many problems after only a few minutes of instruction.

V. PATH PLANNING

A. RATIONALE FOR INCLUDING PATH PLANNING

One of the most difficult times for a sailor is reporting to a new ship. The sailor is under the stress of entering a new job, not knowing anyone else on the ship, being away from his family, and, potentially, being at sea for the first time. In addition, it is very difficult for the new sailor to learn his way around the ship. The uniform construction of the ship makes one area look much like another, so it is difficult to get landmarks to aid in navigation. Quite often, all these factors make reporting to a new ship a very stressful time for young sailors.

In addition, the newly reported sailor usually doesn't help his ship for quite a while after reporting aboard. For many months, all his time is spent trying to get acclimated to his surroundings and learn his job, his watchstation, his chain of command, even his bunk location. All these add up and mean that for the first six months, the new sailor is unable to perform as a full-fledged crew member. Given that most tour lengths are between eighteen and twenty-four months, he is spending one-quarter to one-third of his time onboard at less than full capacity.

However, if the sailor has been able to explore his ship in a virtual world before reporting aboard, it will make the acclimation process less stressful. Instead of being just another unknown, the ship might be the only familiar part of entirely new surroundings if the sailor has spent time in a virtual environment of his ship. This would reduce the amount of stress a newly reported sailor experiences. Since he has spent time in his ship "virtually" prior to reporting aboard, the amount of time spent learning the basics is reduced, and he becomes a contributing member of the crew much faster.

To facilitate this learning process, a feature was added which allows the simulation to demonstrate to the user how to get from his current location to another location in the ship. Unlike the "Translate" feature, where the user is instantaneously transferred to the place he wishes to be, this feature takes the user along the path he must transit to get there. This

allows him to see where he should be going and make mental notes so that he will be able to transit the path again without help. The interface for this feature was described in the Chapter IV, while the algorithm is described below.

B. CREATING A PATH

The basic algorithm for path planning is very simple. When the user selects the "Show path" option, the application must first find what portion of the ship the user is in and then determine how to get from that portion of the ship to the endpoint of the path, the user's destination. How this is done, however, is not so simple.

The number of paths from any point in the ship to any other point in the ship is infinite. If it were possible to take a straight line path from one point to another, this would not be a problem. The path could be created by drawing a line segment between the user's location in three space and the destination's location and traversing that line segment. However, this path would indiscriminately travel through bulkheads, decks and overheads; since the sailor could not follow the path in reality, he would not learn much from the experience.

In order to reduce the problem to a size which might be handled, the number of destinations is reduced to just the major portions of the ship, such as CIC, DC Central, and the Radar Room. However, the remaining problem is still too large to solve easily, because the user might be anywhere when he selects the "Show path" option. To reduce the problem further, the ship is subdivided into small units, each of which is a rectangular cube and has a point in three space associated with it, called a "checkpoint." The bounding volumes are defined so that a user in any portion of the bounding volume can get to its checkpoint in a straight line which a person could transit, i.e., it does not go through any bulkheads, over ladder wells, etc. Figure 25 shows an example of dividing a deck into bounding volumes with checkpoints. It is important to note that while every bounding volume has a checkpoint associated with it, there are checkpoints which are not associated with bounding volumes. During the initialization of the application, data containing the boundaries of each cube and the location of the checkpoints are loaded into a matrix named "paths" from the file

"locationData.dat" located in the "models" subdirectory. Also during initialization, the file "pathData.dat" is read to create an N x N matrix of integers named "nextPlace," where N is the number of checkpoints in the database, whose use will be explained below.
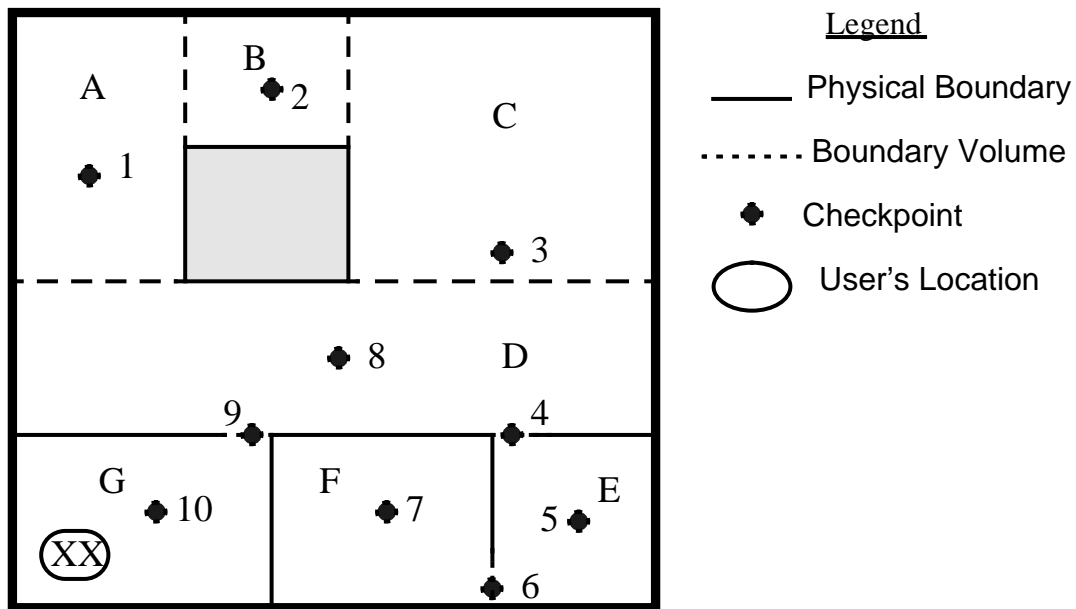
Figur e 25: Example of Bounding V olumes

When the user selects his destination using the "Show path" option, the algorithm first determines the bounding volume the user is in by comparing his location in three space against the boundaries of the cubes stored in "paths." When the application finds the cube which holds the user's position, it then determines where the checkpoint of that cube is. It slowly turns the user's viewpoint so it is facing the checkpoint, and then moves the viewpoint to the checkpoint using small, discrete steps which simulate walking in the virtual environment.

Once the user reaches the first checkpoint, the algorithm uses a table driven solution to solve the problem. At that checkpoint, the application knows at which checkpoint the user is and to which checkpoint he wants to go. Rather than having each checkpoint store a long path in memory containing a path to traverse to any of the destinations, the path from a checkpoint to a destination is given as a series of checkpoints. This series of checkpoints

is stored in the matrix "nextPlace." An example of what this matrix would look like for the

database in Figure 25 is given in Table 2. The program enters the matrix with the ordered

pair (current checkpoint, destination checkpoint) as the indices; the integer located in that

cell of the matrix is the checkpoint to which user must transit to get to his destination. The

application knows the location of each checkpoint and simply transits a line segment

between the checkpoints in small, discrete steps which simulates walking.

	1	2	3	4	5	6	7	8	9	10
1	-1	1	2	3	4	4	6	3	1	9
2	1	-1	2	3	4	5	6	3	3	9
3	2	3	-1	3	4	4	6	3	3	9
4	2	3	4	-1	4	4	6	4	8	9
5	2	3	4	5	-1	5	6	4	8	9
6	2	3	4	6	6	-1	6	4	8	9
7	2	3	4	6	6	7	-1	4	8	9
8	9	3	8	8	4	4	6	-1	8	9
9	9	1	9	8	4	5	6	9	-1	9
10	9	1	9	8	4	4	6	9	10	-1

Table 2: Sample Checkpoint Matrix

For example, using the database in Figure 25 and the matrix in Table 2, assume that a

user at location "XX" wishes to go to a destination located at checkpoint 7. Once the user

selects "Show path to: 7", the application will first compare the user's location to the

boundaries of all the bounding volumes and determine that the user is in volume "G,"

whose checkpoint is checkpoint 10. The application will turn the user's viewpoint so it is

facing checkpoint 10 and transit the user's viewpoint to checkpoint 10. It then enters the

matrix with the ordered pair (10, 7) and finds that the next checkpoint it must go to is

checkpoint 9. It turns the user's viewpoint so it is facing checkpoint nine and transits it to

checkpoint 9. It enters the matrix with the ordered pair (9, 7) and finds the next checkpoint is checkpoint 8. It repeats the process, moving to checkpoints 8, 4, 6 and 7. When it enters the matrix with the pair (7, 7), the value -1 is returned. When a negative value is returned, it means that the user is at his destination and the process is stopped.

C. SUMMARY

Demonstrating paths to users is intended as a method to allow new sailors to navigate their ship. It does this by displaying the path in the virtual world and giving him visual clues that he can apply in reality. The algorithm applies a simple form of artificial intelligence to determine the path required to get the sailor from his current location to his destination.

VI. COLLISION DETECTION

As the participant moves around in the ship walkthrough environment, he may wish to interact with the virtual objects in the synthetic world. Total immersion requires a virtual environment to give the user a feeling of presence, which includes making the images of both the user and the surrounding objects feel solid. For example, the user should not be allowed to pass through walls, and objects should move as expected when manipulated. Such actions require accurate collision detection and response if the simulation is to achieve any degree of realism. However, there may be thousands, even millions of objects in the ship walkthrough system, so a naive algorithm takes an unacceptably long time to check for collisions and other interference as the user moves. This is especially serious in real-time simulation systems, where the issues of interactivity impose fundamental constraints on the system. [ZELT92]

A. PREVIOUS WORK

The problems of contact determination and interference detection have been extensively studied in computer graphics, robotics, computational geometry, and modeling literature for several decades. However, none of the algorithms constructed previously adequately addresses the issue of collision detection in an interactive virtual environment. An interactive, large-scaled ship walkthrough poses a new challenge to the collision detection problem because it requires performance at interactive rates for potentially millions of pairwise tests. Very recently, Hubbard proposed a solution which addresses this problem by an interactive collision detection algorithm that trades accuracy for speed [HUBB93]. However, exact contact determination is required in some applications involving detailed collision response, such as predicting a ball's path as it bounces off walls, so this trade-off is not always acceptable.

Lin showed that accurate, real-time performance could be attained in a large scale model if "temporal and geometric coherence" between frames is implemented. This

method not only speeded up pairwise interference tests, but also reduced the actual number of these tests performed. [LIN93]

The collision detection algorithm employed in the virtual ship walkthrough application is based on Iris Performer's collision detection method. A set of line segments is tested against the scene database. If an intersection occurs between a line segment and anything in the scene, information is returned about the object the segment intersected. This is an excellent algorithm for terrain following and simple collision detection between objects. However, it is a very cumbersome method to try and determine collisions between many objects of different sizes. Obtaining adequate spatial coverage of the differently shaped objects requires many line segments. Silicon Graphics has acknowledged this shortcoming and plans to improve their method of testing for intersections in a future release of Performer. The new method will be similar to Lin's method and will test scene to scene the intersection between polyhedral volumes. [ROHL94]

B. INTERSECTION TESTING

As mentioned above, computing intersections using Performer is based entirely on sets of line segments. The line segments are embedded in a "pfSegSet" structure which holds the origin, length and direction of each line segment. This structure also contains an intersection traversal mask. This mask is a binary number which can be set to different values to selectively determine what type of objects to collide with. The intersection traversal of the visual database is invoked by the application by registering a callback during the initialization phase.

In order to discriminate between different types of objects in the database, the objects in the database contain intersection masks as well. An intersection test occurs with the object only if the bitwise AND of the intersection traversal mask and the intersection mask of the object result in a non-zero value. If the result is non-zero, the line segments in the "pfSegSet" structure are tested against the geometry of the object. If an intersection occurs, vital information concerning the intersection point and intersected object are saved for use

by the application. If the intersection test results in a zero, the traversal continues without traversing any of the objects children.

The object's intersection mask, which is part of each object's data structure, is determined and assigned as the model is loaded during program initiation. The loading callback accomplishes this by analyzing the special fields of an object and placing that object into an array with the correct intersection mask. These arrays are divided according to the particular type of object they contain. Once the loading process is completed, a traversal of the entire visual database is conducted and intersection masks are assigned to objects and their children based upon what the object is.

As alluded to earlier, when an intersection occurs Performer saves vital information about the object it collided with in a "pfHit" structure. Some of the information which is stored in the "pfHit" structure includes the coordinates of the intersection, the normal vector at the point of intersection, the transformation matrix of the object intersected, the node name and a pointer to the object in which the intersection occurred. To extract information from the "pfHit" structure, a query is sent which returns requested information from the "pfHit" structure. By default, the "pfHit" structure returns the nearest object collided with when queried. This can be changed by a discriminator callback.

The intersection process used in this simulation includes terrain or deck following, object collision detection and picking. Even with several types of tests, the intersection process does not the limit frame rate on a multiprocessor machine, and only slightly on a single processor machine. This is because the time required for the intersection testing is comparable to the cull process and is much quicker than the draw processes. Therefore, it was deemed unnecessary to attempt to improve upon Performer's collision detection scheme until it became a limiting factor in the performance of the application.

C.   TYPES OF COLLISION

1.   Deck Collision

In order to simulate the observer walking through the ship environment, an intersection request is conducted each frame to determine the eye point of the observer. A line segment with a length of twenty meters, pointing in the negative Z direction (straight down) from the observer's eye coordinates is loaded into the pfSegSet structure. The length of the line segment is twenty meters since this is the maximum height above an object which can be encountered while walking through the ship. This occurs if the user jumps into the water from the bridge of the ship. By having a ground or deck collision each frame, the correct height of eye can be calculated constantly.

To prevent walking on tables, pumps, piping and other equipment, the traversal intersection mask is assigned a distinct number. As discussed earlier, each object was given an intersection mask during the loading process based upon its type. The bitwise "AND" of the object intersection mask and traversal intersection mask determines if intersection testing should occur. For example, our code contains the following intersection masks:

- #define   DECK_MASK   0x10   //Deck Intersection Mask
- #define   PUMP_MASK   0x02   //Pump Object Intersection Mask
- #define   DECK_COLLIDE   0x10 //Deck Collision Traversal Intersection Mask

A pump has PUMP_MASK as its intersection mask and a deck has DECK_MASK as its intersection mask. The intersection request for deck collision contains DECK_COLLIDE as the traversal intersection mask. Therefore, if the eye point of the user is over a pump, an intersection does not happen with the pump since the bitwise "AND" of the traversal mask and object mask is zero, causing the pump to be pruned from the traversal. On the other hand, the bitwise "AND" of the deck mask and traversal mask is non-zero, which results in an intersection test. If an intersection occurs, information about the deck is loaded into the "pfHit" structure.

To calculate the height of eye of the user, a query is conducted with the "pfHit" structure which returns the intersection coordinate of the object. We use the Z coordinate

of the intersected object to determine at what height to put the user's viewpoint. The user's height of eye is added to the Z coordinate of the intersected object resulting in the correct Z coordinate for the user's viewpoint. If the previous height of eye has changed due to ground collision, meaning that stairs are being traversed or an abrupt change in the deck height has occurred, the user falls to the new height of the collided object.

2.   Object Collision

In order to detect collisions with objects, a set of line segments from the user's position in the direction of the user's motion is loaded into the "pfSegSet" structure prior to each frame. The line segments are protracted out in three directions; one segment sticks straight ahead, while the other two are at forty-five degrees on either side of the direction of motion as shown in Figure 26. There are two such sets of line segments originating from the user's X and Y position, one at the user's height of eye, the other at knee level. Two sets are used to enable intersections to occur with objects such as a tables or pipe hangers which do not reach the user's eye level. The lengths of the line segments protracting to the sides are chosen to simulate the width of a human. The direction of the line segments is reversed if the use is walking backwards.
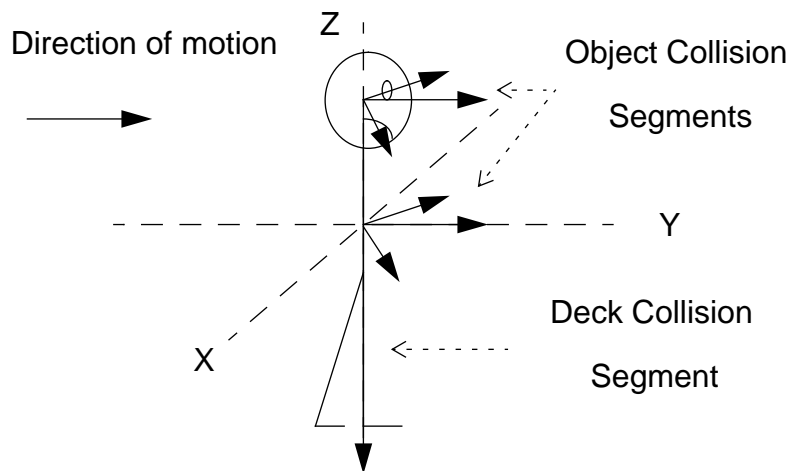
Figur e 26: Collision Segments

The length of the line segment in the direction of motion is calculated each frame by simply calculating the difference between the last view position and the present

position then multiplying by a safety factor of 1.5. This prevents the user from passing through a bulkhead prior to testing it for intersections. If there was no user movement between the previous and current frames, a line segment is generated in the user's eye point direction. The length of the line segment was chosen to be long enough to prevent passing through an object given the maximum speed the user could traverse in an average frame cycle.

When a collision occurs, a global collision flag is immediately set. The application process, which calculates the user's next position, recognizes the collision flag and sets the user's speed to zero. Also, the user's position is set to the last position recorded prior to the collision. This gives the affect of bouncing back off the collided object. To break away from the object collided with, the user must reverse direction or turn away from the object enough to make the line segment generated in the view direction not intersect the object the user collided with.

3.   Picking

Picking is the ability to grab an object with an input device and perform some function with it. In this ship walkthrough, picking is accomplished by depressing the middle mouse button and either the right or left mouse button at the same time. The object picked will be the object at which the mouse cursor is pointing. If wearing the head-mounted display, the object picked will be the object displayed at the cross hairs in the middle of the display.

During the application process, if it is detected that the mouse buttons are depressed to initiate picking, a Performer function called "pfChanPick" which returns the node or object picked. It performs this function in much the same way an intersection request is performed. The inputs to the function include the normalized screen coordinates of the mouse or cross hairs. The function generates a ray which originates from the user's eye point and passes through a point on the near clipping plane generated by the screen

coordinates of the mouse. The objects intersected by the ray are queued into an array of "pfHit" structures.

To obtain information concerning the object picked, a query is made to the "pfHit" structure and the nearest object hit is returned. The "pfHit" structure returned includes a pointer to the node hit, which is used to determine if an object which can be manipulated has been selected. If so, the pointer is used by the program to perform the required manipulations.

This feature is used to manipulate objects in the database, such as doors, valves, and nozzles. To open a door, the right and middle mouse button must be depressed at the same time with the mouse pointing to the door. The pick testing above is performed and a pointer to the door is returned. The application begins to rotate the door in its open direction until it reaches its maximum rotation of ninety degrees or until the mouse buttons are released. To close the door, the left and middle mouse buttons are depressed at the same time, and the opposite motion occurs. The door rotates three degrees every frame while being picked.

The operation of valves is similar to the operation of doors as far as the method used to open and shut the valves. Valves undergo translation movements as well as rotation movements. When opening a valve the valve stem rises and the valve hand-wheel rotates in either the counter-clockwise direction; the opposite occurs when the valve is shutting.

Picking is used for several other functions in this ship walkthrough. When an object which cannot be manipulated is picked, a the hypertext display is put onto the screen and information about the object is displayed. Actions such as opening and closing fire hose nozzles, turning off and on ventilation fans, and activating the halon control system. The algorithm distinguishes between the types of nodes selected by using the intersection mask each object possesses and an array of pointers to the objects which were generated during the database loading process discussed above and in Chapter III. The pseudocode displayed

in Figure 27 explains the algorithm for determining what operations to perform on the object picked.

```
// Object determination and operation
mask = pfGetNodeTravMask(ObjectPicked);
switch (mask)
{
 case INFO_MASK:
   index = findSelectedNode(ObjectPicked, mask);
   DisplayText(ArrayInfoNodePointers[index]);
   break;
 case MANIP_MASK:
   index = findSelectedNode(ObjectPicked, mask);
   ManipulateObject(ArrayManipNodePointers[index]);
   break;
 default:
   break;
}
```

Figur e 27: Pseudocode for Object Mask Operation

D. SUMMARY

Collision detection is a key concept in this application. It aids the navigation algorithm for walking, simulates realism by stopping the user when running in to objects, and enables object manipulation through the "picking" algorithm.

Although there are more efficient algorithms to conduct collision detection, performance has not been inhibited with the methods employed in this thesis. Since the collision detection scheme traverses the hierarchical database, only performing intersection test on objects which pass the intersection masks test, and since the intersection test only checks seven line segments against the geometry each frame, the present method has been found to be satisfactory. As more interaction is added to this thesis between objects, requiring many more line segments to test against the geometry, performance will certainly suffer. Therefore, additional work will be required to improve the performance of the collision detection algorithm as more interaction between objects occurs and more precise collision detection becomes necessary.

VII. POTENTIALLY VISIBLE SETS

A. THEORY

One of the traditional impediments to creating large scale architectural walkthroughs has been the time required to determine which portion of the database to display on the screen. This is normally done in two steps, during the cull process and the draw process, which are discussed in Chapter II. Methods to maximize the performance of these two processes are discussed in Chapter III, specifically the user of hierarchical data structures. However, once a visual database passes a certain size, even these methods result in fairly poor performance. There is simply too much data to cull quickly, and too many objects left in the display list by the cull process to draw quickly.

A simple assumption based upon this premise is that by reducing the amount the application needs to cull, the speed of the application is increased. The problem is that by removing portions of the database from culling consideration, it is possible, even likely, that portions of the database which should be visible to the user will not be rendered, thus reducing the realism of the simulation.

In addition, once the parts of the database not in the viewing frustum are culled from consideration to be drawn, most of the remaining polygons will not be drawn. This is because they will be occluded from the user's view by other portions of the database. For example, if a user is one foot away from the outside wall of a building, he only sees the wall in front of him. However, most of the interior of the building has not been culled from consideration to be drawn since it is in the viewing frustum, and the draw process must spend time deciding that those polygons should not be drawn.

To overcome this encumbrance and increase the speed of an application while maintaining a high fidelity, potentially visible sets (PVS) are used. This theory, first espoused by Airey in his Ph.D. thesis at the University of North Carolina, Chapel Hill [AIRE90B], and expounded upon by Funkhouser at University of California, Berkeley [FUNK92][FUNK94], has been primarily used for building walkthroughs. PVS divides the

database spatially into various volumes. Each volume, called a **cell**, has a limited number of areas which can be viewed by a user in that cell. The number of potentially visible areas are limited due to the presence of opaque objects, such as walls, floors, ceilings, bookcases, etc. Breaks in these objects, such as doors and windows, are called **portals**. These allow the user to see into cells other than the one he is in, and sight lines through more than one portal permit the user to view a potentially unlimited number of cells. However, this is extremely unlikely, and normally the number of other cells potentially visible from one cell is quite limited. Because of this, it is not necessary to perform culling operations on all the portions of the database, but only those few which are potentially visible from the cell the user is currently in. Also, once the database is culled, the draw process is greatly speeded up because there are fewer polygons receiving consideration to be drawn. For example, Figure 28 shows an architectural database consisting of several rooms, which are identified by capitol letters. While in one of these rooms, it is impossible to see all the others; in fact, in some of them, it is only possible to see into one other room. Because of this, it is unnecessary to examine the entire database to determine what is visible. For example, consider a user at location "1" in room "G", looking in the direction of the arrow. If potentially visible sets were not used, rooms "A" and "B" would be culled during the cull routine and the draw process would decide not to display areas "C", "D", and "E", since, even though they are in the view frustum, they are hidden behind walls. However, if potentially visible sets are used, the cull process would not have to spend time considering rooms "A" and "B", since they can never be viewed from room "F", nor would the draw process need to decide that areas "C", "D", and "E" cannot be seen, for the same reason. Because of these "shortcuts", the frame rate while displaying this database would be much greater using PVS than otherwise.

Buildings lend themselves exceedingly well to this sort of division, since most buildings are inherently divided into cells by their walls, floors and ceilings. In addition, the portals are normally quite limited, causing the number of potentially visible sets from each cell to be very low, reducing the culling and draw loads on the application. Other

Location -> PVS

A -> C,D,E,F
B -> D,E,F
C -> A,D
D -> A,B,C,E,F
E -> A,B,C,D
F -> A,B,D,G
G -> F
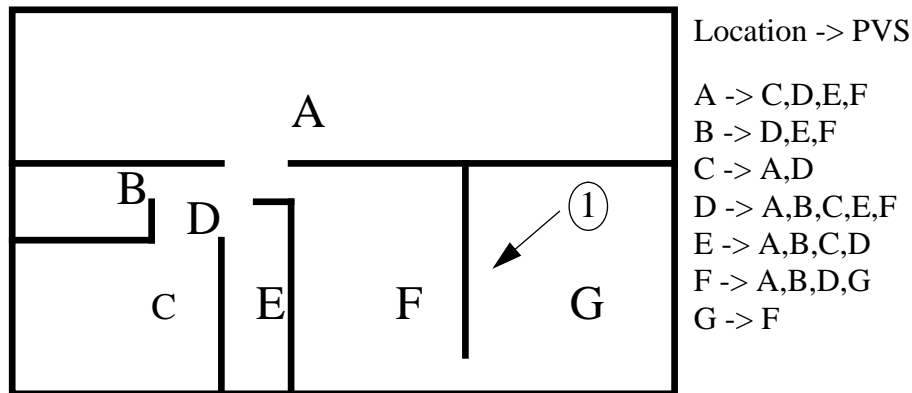
Figur e 28: Building Divided into Cells with Potentially V isible Sets Listed

forms of simulations, such as flight simulators or wide-area wargaming simulations, do not benefit as greatly, since a majority of the database is potentially visible from any point in the simulation.

B.   RATIONALE FOR USE OF PVS IN THE SHIPBOARD VET

Even having organized the database effectually using a hierarchical data structure as described in Chapter III and using Performer's efficient algorithms to cull and draw the database, a database the size of Antares would significantly slow the application. As discussed above, PVS are especially effective when the database contains a large number of occluding objects. A ship is divided much more than normal buildings for a variety of reasons: maintaining watertight integrity, limited space, and limiting battle damage being paramount among them. Therefore, since a ship lends itself so well to spatial subdivision using PVS, which implies a correspondingly large increase in the frame rate, PVS is used to increase the speed of the application.

One disadvantage of creating the database from scratch rather than the actual ship data was that the majority of compartments modeled were much larger than compartments of actual ships. Since this thesis required modeling as large a portion of the ship as possible in a limited time period, detail suffered. Therefore, areas of the ship which would have actually been subdivided into several smaller compartments were left as larger, non-

divided spaces. A prime example of this is the area on the Damage Control Deck. Although Damage Control Central and the Hull Technician Shop were accurately sized, the other spaces on this deck were much larger than those actually found on ship. These spaces would have been divided into many smaller spaces, such as fan rooms, repair lockers, offices, etc. If all the spaces had been so further divided, this would have reduced the volume which would have been potentially visible from each cell of the ship. This means whatever increase in performance is gained by using PVS can be expected to have been greater if the model had been more accurately subdivided.

C. DETERMINING CELLS AND CELL VISIBILITY

Many of the people who performed the early work in PVS created involved algorithms to determine the boundaries of each cell and which cells are potentially visible from that cell. Both [AIRE90A] and [FUNK94] went to great lengths to create computerized methods of breaking the model into cells and determining which portions of the database are visible from each cell. These involve determining sight lines through portals, creating adjacency graphs for the entire database, recursive searches through the database, and many other methods to divide the database as efficiently as possible.

These advanced methods were determined to be beyond the scope of this thesis, as there simply was not time to build such a computer program and make significant progress on the final application in the allotted time. Instead, an extremely simplistic approach was used, which nonetheless demonstrated the efficiency of the PVS for shipboard applications and was sufficient for this simulation. The database was divided not by computing visible sight lines through portals, but instead by dividing the database along natural boundaries, such as bulkheads. Each cell was a rectangular cube, which simplified determining which cell the user was in.

Creating a list of PVS for each cell was similarly simplified. It consisted of the modeler predicting which cells were visible from each of the other cells, implementing that prediction and testing it by running the application. Any deficiencies were noted and

corrected, the PVS data file updated, and the new prediction tested in the same way. Although this method was tedious, it is adequate for this application, since the number of PVS is limited and the method of creating cells is simplistic.

However, these simplistic methods only served to reinforce the importance of the methods Airey and Funkhouser used. It is obvious how dividing the database into even more PVS would increase performance, yet it is extremely time consuming to do this by hand. Even for the relatively simple model used in this thesis, it was impossible manually to predict with a high degree of certainty what will be visible from each cell. Although the cost of this inaccuracy was relatively low, it was only acceptable since the number of cells in this model is quite limited compared to what is required for a highly realistic model of an entire ship. In that case, the time spent changing the PVS data would be prohibitive, and it would be essential to have a method of automating this process.

D.   MODELLING TO ENHANCE USING POTENTIALLY VISIBLE SETS

1.   Modifying the Model

The hierarchical data structure discussed in the Chapter III facilitated using potentially visible sets. As discussed above, the model was not divided as finely as possible, but instead cells were created by dividing the database along inherent boundaries. Since the database had been previously divided spatially to enhance the cull process, it only required minor changes to modify it to support potentially visible sets.

These minor changes consisted of three operations. The first was moving some objects around in the hierarchy so that similar objects which are visible from the same areas are under the same group. This reduces the number of groups which are visible from each cell, which simplifies the algorithm and search process. The second operation was dividing a single group into several so that the number of polygons which were not potentially visible from the user's position, but were nonetheless retained by the PVS algorithm, was minimized. The final was running the application and looking for areas where polygons were included as potentially visible when they were not, and more importantly, looking for

polygons which should be visible but were not because of the PVS data. After this was done, the process was repeated, until the database was fairly tightly divided. It is important to note that this process does not create cells, but rather groups geometry. The common property is that all of the geometry in a group is potentially visible from the same cells, or at least to the extent of the algorithm.

2.   Embedding PVS Data into the Database

Once the database had been suitably divided, the next task was to include information in it so that the application would be able to use the PVS. Each group which contained a single PVS is given an index, which is encoded in the database using the same methods used to embed other information into the model, discussed in Chapter III. Before the model is loaded, the loading callback creates a matrix of node pointers, named "PVSMatrix." As the loader is loading nodes from the model, it does not add the geometry to the scene. Instead, the loading callback checks each group to determine if it is the root node of a PVS. If so, a pointer to that node is stored in "PVSMatrix", under the index which is embedded in that group. Thus, after the loading process, the scene is still empty and a matrix consisting of pointers to the root nodes of all the PVS has been created.

E.   THE PVS ALGORITHM AND ITS IMPLEMENTATION

1.   Data File Makeup and PVS Initialization

After the model has been loaded and the matrix of node pointers has been created, the PVS system is set up to manipulate the data encoded in the model. The data required to utilize this information is included in a data file named "pvsData.dat" in the "models" directory. Each line of this file contains the data for single cell, which is stored in a matrix named "PVSinfoMatrix" as it is read. This cell data consists of an integer which serves as an index, six floats which are the minimum and maximum boundary values in three dimensions of the cell, and an unspecified number of integers, which are the indices of the sets which are visible from that cell in ascending order. It should be noted that even

though the database was primarily divided along compartment lines, the cell information in "PVSinfoMatrix" and the set information in "PVSMatrix" do not necessarily correspond. Thus, the set numbered fifteen might not be visible from the cell numbered fifteen.

After the information has been loaded from the data file, the PVS system is initialized. This consists of finding what cell the user is initially in by looping through the values in "PVSinfoMatrix" and looking for a cell whose minimum and maximum boundary values bracket the user's initial position. Once such a cell is found, the application realizes that is the cell that the user is in and acts accordingly. It takes the indices stored for that cell in "PVSinfoMatrix" and loads the nodes with those indices from "PVSMatrix" into the scene. This results in the correct sets being loaded for the user's initial cell.

2.   Basic Algorithm

Once the PVS system has been initialized, the system will automatically handle updating which PVS are visible based upon the user's location. This is performed in the function "updatePVS", which is called in the application process previous to "pfFrame." By calling "updatePVS" as part of the application process, the frame rate of the simulation is not affected since the application process is not the limiting process in the system. Thus, whatever gains PVS creates come without adding any actual overhead.

The basic algorithm to determine which PVS to display was streamlined by the simplicity of the division of the database. Since each of the cells is a cube, determining if the user has left the cell is only a matter of comparing the user's position in three dimensions to the boundary values of the cell which he was in the last frame. If the boundary values of that cell still bracket his position, nothing more need be done and the function returns.

If, however, the user's position is outside of any of the cell's boundary values, the algorithm searches "PVSinfoMatrix" to determine what new cell the user has entered. If the user's position does not meet the boundary values of any of the cells in

"PVSinfoMatrix", the PVS manager will load all the sets to the scene. The cell values were constructed so that this should never happen; however, most bugs "should never happen." Because a degradation in performance was considered preferable to confusing the user by having geometry which is visible not rendered, this feature was added. When the user reaches a volume recognized by the PVS manager, all the superfluous geometry is removed and the application continues at the normal, high frame rate.

Once PVS manager determines in which cell the user is now, it must determine what geometry to add to the scene. It is impossible to simply add the geometry in the new cell to the scene and maintain efficiency. This is because it is likely that two adjacent cells, for example cell "A" and cell "B", share many of the same PVS. If, as the user leaves cell "A" and enters cell "B", all the sets visible from cell "B" are simply added to the scene, it results in the scene containing two copies of the sets that "A" and "B" have in common. If the application runs for any length of time, the number of copies of each piece of geometry in the scene becomes quite large. This increases the culling and drawing load of the application, to the point where it soon negates the gains made by using PVS and actually makes performance worse than if PVS is not used. It is possible to avoid this problem by removing all the geometry currently in the scene before adding all the PVS of the new cell; however, this operation has a fairly high overhead. While this penalty is not noticed with a small database like the one used in this application, it becomes quite a hinderance in an application using a larger database.

A rather complex algorithm is used to update the list of sets being displayed to avoid this potential pitfall. Figure 29 displays the algorithm in pseudocode. The algorithm starts by setting currentlyVisible to the index of the first PVS for the cell the user was previously in and nextVisible to the index of the first PVS displayed in the cell in which the user is now. If then compares these two values at the line marked by (e) in Figure 29. Because the indices were in the data file in ascending order, if currentlyVisible is less than the nextVisible, it means that the PVS of the previous cell with the index of currentlyVisible is not displayed, and that cell is removed from the scene. currentlyVisible

74

is then set to the value of the next index visible in the previous cell and the process loops. If, instead, the "if" statement at (1) is not true, it again compares the values of nextVisible and currentlyVisible at (2), and if nextVisible is less than the value of currentlyVisible, it means that the node indexed by nextVisible is not currently being displayed and should be, so it is added to the scene, and nextVisible is set to the index of the next PVS of the new cell. If both (1) and (2) are false, the same PVS is being indexed by both variables, which means that PVS is currently being displayed and needs to be displayed. Therefore, the else at (3) is executed, and both variables are merely incremented to the next index of their respective cell. If all the PVS from the new cell have been added to the scene, it means that all the PVS remaining from the old cell need to be removed from the scene. On the other hand, if all the PVS from the old cell have been removed, it means that all the remaining PVS from the new cell can be added to the scene. These cases are handled by the "if" statements at (4) and (5).

3.   Efficiency of the Algorithm

The method of computing which cell the user is in was streamlined by the simplicity of the division of the database. Since each of the cells is a cube, determining which cube the user is in is only a matter of performing six floating point comparisons for each of the cells in the database. On average this will take 6*N/2 computations, which is (O(N)), where N is the number of cells into which the database is divided, which is twenty for this model. However, by making the algorithm first check those cells the user is likely to be in, this average can be improved, to what extent depends on the nature of the application. For example, in an engineering training application which uses the same model, where most of the user's time is spent in the engine room and two auxiliary spaces, putting those spaces in the search list first will greatly reduce the search time. Although the result is still O(N), there is still a threefold increase in performance, dropping the average from sixty computations to slightly greater than eighteen computations.

```
currentlyVisible = index of f          irst PVS for old cell
nextVisible = index of f        irst PVS for new cell
loop{
/* If this f          irst "if" is true, it means that the node indexed
by currentlyVisible is not visible in the new PVS, so it
should be removed from the scene */
          if (currentlyVisible < nextVisible)                          (1)
                    remove node indexed by currentlyVisible from scene
                    currentlyVisible = index of next PVS for old cell
          else
/* If this next "if" is true, it means that the node indexed
by nextVisible is not visible in the current PVS, so it should
be added to the scene */
          if(currentlyVisible > nextVisible)                           (2)
                    add node indexed by nextVisible to scene
                    nextVisible = index of nextPVS for new cell
/* If this else is reached, it means that the node indexed by
nextPVS is already visible in the current PVS, so nothing need
be done other than increment counters */
          else                                                         (3)
                    nextVisible = index of nextPVS for new cell
                    currentlyVisible = index of next PVS for old
cell
                    if(no more PVS for new cell)                       (4)
                              delete the remaining PVS of old cell
and exit loop
                    if(no more PVS for old cell)                       (5)
                              add the remaining PVS of new cell and
exit loop
}end loop
```

Figur e 29: Pseudo Code for PVS Geometry Replacement Algorithm

Once the application has determined which cell the user is in, it is only necessary to perform six comparisons to check if the user is still in the same cell. If the user has left the cell he was previously in, the process of determining which cell the user is in is repeated.

F.   RESULTS OF USING PVS

There are several methods to measure the effects of using PVS to increase the speed of applications. One common method is to examine the reduction in the numbers of objects and polygons visible to each cell compared to those in the entire database. The entire ship contains 911 objects and 22,840 polygons. The minimum reduction in both the number of objects and the number of polygons occurs in the Engine Room Upper Level cell. This cell

has 385 objects and 7644 polygons, which produces a 57.7% reduction in the number of objects and a 66.5% reduction in the number of polygons. This is expected, since from this cell it is possible to see all of the engine room and Engine Room Landing. The maximum reduction occurs in the Operations Office cell, which is also as expected, since the only other place visible from the Operations Office is the Ops Landing. The Operations Office cell contains 58 objects and 571 polygons, which is a 93.6% reduction in objects and a 97.5% reduction in polygons from the entire model. The average cell consists of 199.2 objects and 3381.7 polygons, which gives a very impressive reduction of 78.1% and 85.2% from the entire model respectively. Further discussion of object and polygon reduction, in addition to data for the entire ship, is included in Appendix B.

However, while this method shows how much the database is reduced by using PVS, this is not the true goal of PVS, and therefore should not be used as the true measure of its effectiveness. The desired result of PVS is to increase the frame rate of the application, and this is the only result which truly matters. Polygon reduction merely gives an indication of how well the algorithm will improve performance. For example, a method of PVS which reduces the average polygon count by 95%, but requires so much overhead that the frame rate increases only 15%, it is not as desirable as a method which reduces the database only 25%, but increases frame rate by 60%.

By this standard, the PVS algorithm does extremely well. It increased the average frame rate on the Reality Engine 2 by 48.1%, the Reality Engine 1 by 42.7%, and the Indigo 2 Extreme by 63.6%. In most cases, the application ran as fast on the Reality Engine I using PVS as it did on the Reality Engine 2 without PVS; the average was just one frame per second faster on the RE2. This means that PVS effectively translated the performance of a machine which will be available soon for $29, 000 to that of a machine five times the price. The increase in performance is also true of the Indigo 2 as compared to the Reality Engine 1. What is also important is that using PVS lifts the frame rate of the RE1 from unacceptable to acceptable, using Airey's assertion that six frames per second is the minimum acceptable for an immersive walkthrough [AIRE90A]. Using PVS on the RE1

raises the average frame rate from 5.5 fps to 9.6; in addition, it raises the average minimum from a grossly unacceptable 3.2 fps to 7.2 fps, still fast enough for immersion. Indeed, with PVS the Indigo 2 Extreme is only 0.5 fps from meeting Airey's minimum. When examining this data, it is also important to remember that, as discussed in Chapter III, the increase in performance due to PVS would be much greater using the model for an actual ship. The methodology used for this experiment and the data taken are contained in Appendix B.

One other point of interest is the difference in the intersection process between the PVS version and the version without PVS. While the time required for the cull and the draw process went up as expected, approximately doubling in the non-PVS version, the time required for the intersection process went up by approximately a factor of five. The intersection process still did not limit frame rate, as the time for the draw process greatly exceeded it, but the intersection process was now significantly longer than the cull process. The reason for this is the nature of the intersection testing. The cull process can quickly discard branches of the tree if no portion of any object in that branch is in the view frustum, and the database is designed to make this as easy as possible. However, the intersection mask of a node is the bitwise "OR" of all its children and their descendants. Therefore, the intersection process has to descend each branch that has any object with an intersection mask which meets its mask. Since the database is not grouped by intersection masks, the intersection process is more likely to have to descend a branch deeply before it can be discarded. The effect of PVS greatly reducing the number of branches in the database is noticed most in the intersection process, since it has to traverse deeper into the branches that are present.

G. SUMMARY

The visual database used for the shipboard VET is too large to be rendered in real time by normal methods, except on an extremely expensive computer. Both the average and minimum frame rates are too low to present the user with any sort of illusion of immersion.

To correct this, a simple form of PVS was designed and implemented. It increased the frame rates to levels which make the user feel he is actually on a ship while still running the application on a machine whose price makes it possible to put several on a Navy ship.

VIII. ENVIRONMENTAL EFFECTS

The environment on board a Navy ship can change drastically during a casualty such as a fire or steam leak. In order for Navy ships to be able to maintain their warfighting capability while sustaining such casualties, the ship's personnel must be highly trained to react immediately and correctly. To achieve this type of response, these personnel receive training at damage control simulators prior to reporting aboard a ship. At these simulators, each individual is required to enter a space that is engulfed in flames and extinguish the fire. This type of training is conducted to effectively simulate the environment which results when a fire occurs on board a ship so that these individuals will be better prepared to fight the actual casualty. By the same token, it is desired that this virtual shipboard trainer effectively simulates the environmental changes which occur when casualties such as a fire or steam leak occur in order that the sailors will be better prepared to fight the actual casualty.

Realistic simulation of the changes which occur in the surrounding environment of the virtual ship due to casualties and the user's actions is the goal for simulating environmental effects. For example, when using a fire hose to put out a fire, the effect of the water hitting and spraying off objects must be simulated, or the realism of the simulation is degraded and the effect on the trainee is degraded. When a steam leak occurs, the spaces in which the steam leak occurred should fill up with steam and reduce visibility, or the simulation will not adequately prepare the trainee for the actual casualty.

Modeling objects such as fire, smoke and water which do not have well defined shapes and surfaces using traditional graphics techniques is difficult. The most commonly used graphics primitives consists of polygons, patches and surfaces, all of which have very distinct and discrete edges. The surfaces of the fuzzy objects are irregular, ill-defined with dynamic surfaces, so traditional primitives are difficult to use. It is the dynamic and fluid property of the fuzzy systems that is desirable and needs to be preserved in the rendered object. Two methods are used in this thesis to preserve these properties: textured polygons

and particle systems. A discussion of both methods as well as simulating the effects in real-time casualty scenarios follows.

A. PREVIOUS WORK

The conceptual ideas in which our fuzzy object paradigms are based were first introduced by Reeves in [REEV83] and Gardner in [GARD85] [GARD92]. Their models are based on physically based equations which are too long to be used in a real-time system.

Reeves presented a method for modeling fuzzy objects using a technique called particle systems in [REEV83]. A particle system differs from an object represented using traditional image synthesis techniques. First, the object is represented as a set of randomly placed primitive particles within a bounded volume rather than by a set of primitive surfaces. Second, the particles are not static but change form, move and die over time.

Gardner presented a method to generate realistic smoke and clouds in [GARD85] and [GARD92]. The models he presented used ellipsoids that are covered using a texture derived as a function of the transmittance of transparency each ellipsoid should possess. The transmittance of transparency varies from the center of the object to the edges of the ellipse as a Gaussian function. The overall effect is realized when a series of textured ellipsoids are generated and subsequently translated as a function of ambient wind conditions.

Environmental effects based on the work of Reeves and Gardner have been produced more recently by Corbin [CORB93] and Watt [WATT94]. Both greatly simplified the computational complexity of the methods introduced by Reeves and Gardner so that the effects could be realized using a real time system.

B. TEXTURED POLYGONS

Textured polygons are used to provide the effects for fire, smoke and water spray in the virtual shipboard trainer. Using traditional graphics primitives, it is difficult to realistically simulate these effects due to their dynamic nature. However, a set of textured polygons which are blended correctly with the surrounding environment and move naturally can

provide a realistic simulation of the dynamic object. Therefore, in the implementation of fire, smoke and water spray, sets of moving textured polygons which are blended with the surrounding environment are used. In implementing fire and smoke, the Performer software utility library was employed which provides smoke and fire data structures and functions. The fire and smoke effect generated with this program is shown in Figure 30. A discussion of each dynamic object follows.
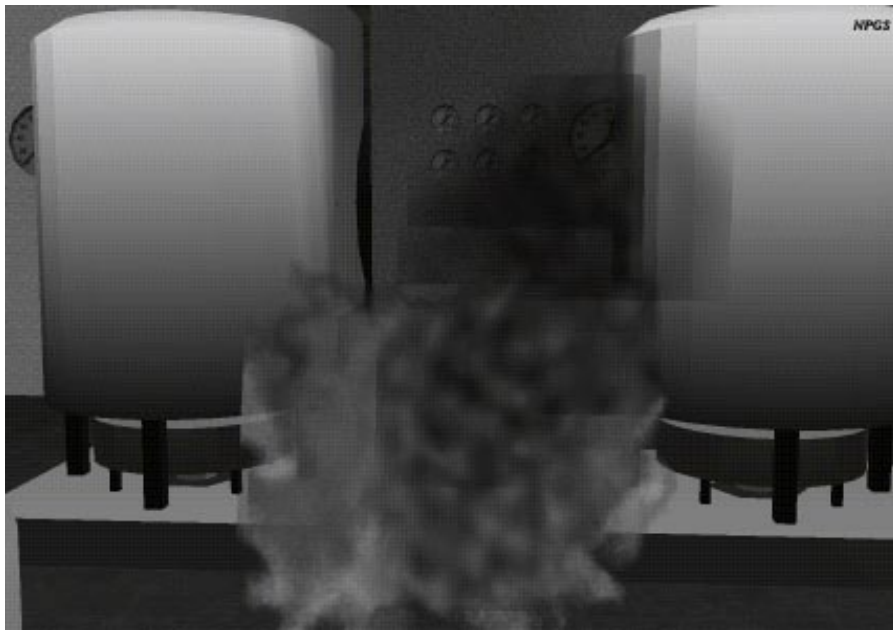
Figur e 30: Fir e and Smoke, Antar es Engine Room

1.   Fire

Simulating a fire in the virtual ship is accomplished with the aid of the Performer utility library. Performer's fire consists of a group of semi-transparent polygons which are textured with an image of a raging fire. In order to make these polygons look like a realistic fire, a utility library function is called which creates a new "pfuSmoke" structure. This structure is a set of attributes which define particular qualities of the fire. Part of the "pfuSmoke" data structure consists of an array of thirty-two "puff" structures. These define the vector coordinates, texture coordinates, start times, speed, direction, radius, and

transparency for each textured polygon. The vector coordinates of each puff are randomly determined within a bounding area surrounding the central location, or origin of the fire. Each "puff" structure is drawn during every frame, and since each "puff" has a different speed and initial location, it is sufficiently random to produce a dynamic effect. The transparency of a puffs increases the further it is from the origin, causing it to blend into the surrounding environment. The "puff", once it travels to the fire boundary outer limits, is reinitialized to start it's sequence again with a new set of attributes.

To start a fire in the virtual ship, the Performer utility library is initialized with the origin of the fire, its initial radius and the user's view position in the virtual ship. This information is passed to the utility library each frame prior to the draw process. To enable the fire to grow, a time dependent growth function is used which increases the radius of the fire as time elapses as long as the user is not extinguishing the fire by either a vari-nozzle or the halon activation system. When this occurs, the radius decreases depending on the type of extinguishing agent applied to the fire and the time it is applied. The user's location is required each frame because the puff is a set of polygons and not a three-dimensional object. Therefore, the polygons must face the viewer at all times to provide a realistic effect.

2.   Smoke

Smoke is generated by the same Performer utility library. The are only two noticeable differences. The first and most obvious is that the texture applied to the polygons is an image of smoke. The other difference is that the smoke puffs rise vertically at quicker speeds, while the fire polygons move slower and remain close to the height of the fire's origin. The smoke radius is also a time dependent function similar to the fire function.

3.   Water Spray

The water spray is a set of six textured polygons connected in the center along one axis and aligned every sixty degrees about the axis. The water spray is loaded into the application program as a dynamic coordinate system (DCS) node, which enables it to be easily sized, translated and rotated. When the fire hose nozzle is opened, a collision

detection algorithm is initiated during the intersection callback process. Four line segments are sent out from the nozzle along the path of the water. The line segments which intersects with the scene geometry closest to the nozzle returns the coordinates of the collision. The textured polygons which make up the water spray are then randomly rotated about the intersected coordinate to provide the effect of water spraying off objects.

C. PARTICLE SYSTEMS

A particle system is used to provide the graphical simulation of a steam leak and oil spray in the virtual ship trainer. An oil spray and steam leak consists of many particles originating from the same location and traveling in random directions at random speeds within a bounded volume. A particle system possesses the attributes to simulate this motion as well as delivering it through the graphics pipeline with essentially no degradation in performance. Examples of the steam leak and oil spray generated in the virtual ship are shown in Figure 31 and Figure 32, respectively.

Particle systems have advantages over objects generated using traditional image synthesis techniques. The first being that it is much simpler to represent than a surface object because orientation is normally not a concern. Hence, the computation time for each primitive is reduced allowing objects made of more primitives to be rendered in the same time period, resulting is a more realistic simulation for essentially the same price. Second, the model definition for translation of the particles is procedural, usually based on physical laws, controlled by random numbers, allowing the modeler to adjust the level of detail to fit the specific situation. [CORB93]

In order to generate images quickly, light points were chosen to represent the particles since it is a very simple structure consisting of only a few primitives. A light point is basically a group of colored pixels which can be placed in the virtual world, where the number of pixels depends on the size you make it and the color depends on the RGB value assigned to it. The particle system can contain up to two thousand light points or particles before system frame rate is affected.
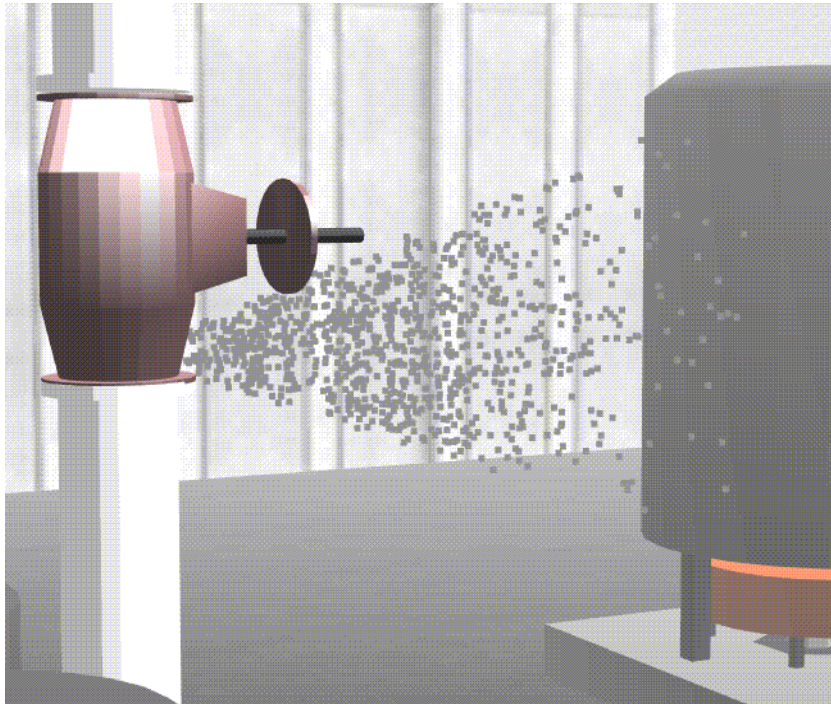
Figur e 31: Steam Particle System
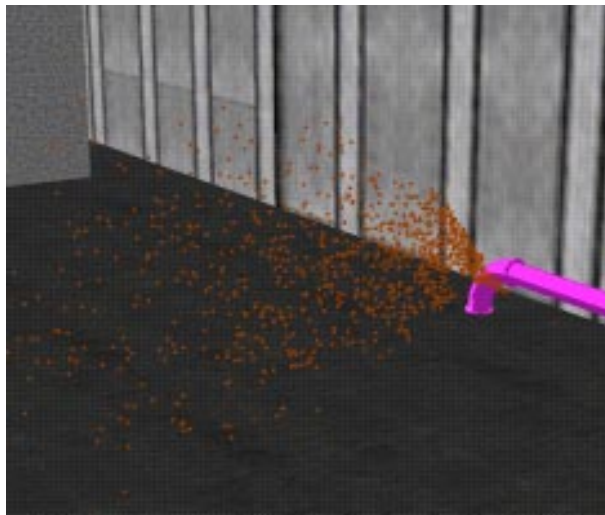
Figur e 32: Oil Spray Particle System

The Performer software embeds light points in a structure called "pfLightPoints," which is a type of "pfNode" in the Performer data structure. A "pfLightPoint" node can

contain any number of light points as children. The children or light points share color and size attributes, yet can be placed in different locations in the virtual environment. By random placement of the light points in a bounded volume, the particle system is generated. The advantage of this code is that it is sufficiently generic that it can be used to create a wide variety of casualties, such as condensate rupture, seawater leak, feed rupture, flooding, etc. The two which are implemented in the shipboard VET are a steam leak and fuel oil spray.

1.   Steam Leak Implementation

Random placement of the particles in the bounded volume does not produce the effect of spray emanating from a source and traveling at an initial velocity a given distance, which is the effect desired for a steam leak. In order to produce this effect each particle is initialized to originate from the origin of the leak, with an initial velocity vector as shown in Figure 33. The initial velocity vector direction is calculated by randomly choosing a target coordinate at the far bounds of the bounded volume and subtracting the origin coordinate. The velocity vector's magnitude is the length of the vector multiplied by a random number between 0.2 and 1.0. The minimum random velocity multiplication factor of 0.2 was chosen to prevent a build up of an excessive amount of slow particles. The particle's location is updated each frame to its new position based on its initial velocity vector. If the new position is beyond the limits of the bounded volume, the particle disappears and is reinitialized to the origin with a new random velocity vector.

2.   Oil Spray Implementation

An oil spray differs from a steam leak in its dynamic motion through the environment. A steam particle, after exiting the steam source with an initial velocity created by the pressure of the steam source, travels a small distance before it becomes intermixed with the atmosphere and blends into it or condenses. On the other hand, an oil particle will travel in the environment until it hits an object. The oil particle takes on a ballistic motion through the atmosphere as gravitational and drag forces affect its velocity vector.

Therefore, the oil spray implementation is exactly like the steam leak implementation with some additional physical based forces imposed on the algorithm. The path of an oil particle is illustrated in Figure 34.
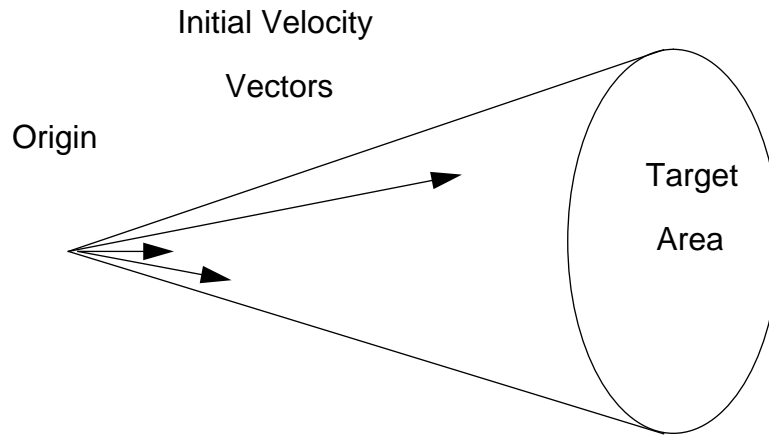
Figur e 33: Particle Initialization
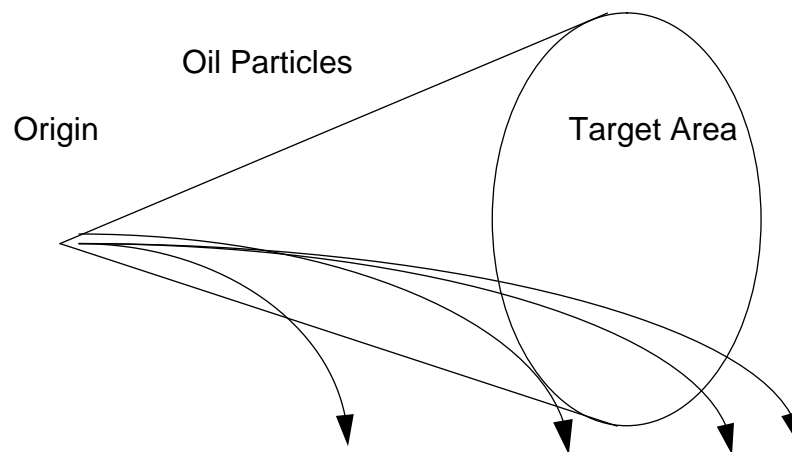
Figur e 34: Oil Particle Path

Like the steam leak, the oil particles originate from the same location with an initial random velocity vector. The oil particle velocity vector is calculated using the same method as the steam leak velocity vector. During its flight the oil particles undergo

gravitational acceleration which cause the particles to fall toward the deck. Unlike the steam particle, the oil particle continues to travel beyond its volumetric bounds until it hits an object to add realism.

The steam leak and oil spray particle systems are initialized and updated using the same procedural call. The physical based motion is applied to the particles during the update stage only if a variable is set to true in the procedural call. The pseudocode for the particle systems is provided in Figure 35.

D. ATMOSPHERIC EFFECTS

The atmosphere on board a ship can change dramatically when a fire or steam leak occurs due to the ship's confined spaces with limited ventilation. To accurately simulate this, fog functions from the Performer software tool kit are used to obscure the user's vision in the virtual world.

The Performer's "pfFog" data structure can be modified to support the illusion of being in a smoke filled or steam filled compartment. It accomplishes this by controlling the hardware fog function to blend each pixel color on the screen with the color of the fog. The amount of blending is dependent on the range of the fog. [SGI94]

The intensity of the smoke and steam in a compartment is a time dependent function. The longer period of time the fire burns or the longer period of time the steam leak continues, the greater the visual obscurity should be. When a fire or a steam leak is initiated, the fog function is called with the color attribute of the obscurity and range attribute of thirty meters. With each frame, the distance of the fog decreases until it reaches a minimum of five meters.

Intensity of the smoke or steam is less in a compartment adjacent to the casualty than in the compartment where the casualty is occurring. This is factored into the obscurity algorithm by changing the range of the fog to correspond to the expected obscurity intensity level in each adjacent compartment. The smoke or steam obscurity will dissipate after the

```
//Particle Initialization
void particleSetUp(Particle, VelocityArray, TargetArray, NumParticles, color, ori-
gin, size, direction, radius, length)
{
    //Figure out bounding volume for particle system
    MakeConeFromOrigin(length, radius, direction, origin);

    //Randomly fill cone target area (circle at end of cone)
    for (i = 0; i < (2*numParticles); i++)
    {
        spacing = generateRandNum();
        TargetArray[i] = placeTarget(spacing);
    }

    //Initialize all particle positions and initial velocities
    for (i = 0; i < numParticles; i++)
    {
        index = generateRandNum(); // 0.2...1
        VelocityArray[i] = TargetArray[index] - origin; // calculate velocity
        Particle[i].pos = origin;
    }
}

//Move Particles (called each frame)
void moveParticles(Particles, targetArray, VelocityArray, numParticles, origin,
length, phsicallyBased)
{
    //, if physically based apply gravitational force to particle
    if (physicallyBased)
    {
        for (i = 0; i < numParticles, i++)
        {
            oldPos = Particle[i].pos;
            Particle[i].pos = oldPos + VelocityArray[i] - Gravity;

            // Check to see if particle hit ground and if so reinitialize it
            if (Particle.posZ < GroundHeight)
                Reinitialize(Particle[i]);
        }
    }
    else
    {//particle travels with initial velocity to limits of volumetric bounds
        for (i = 0; i < numparticles; i++)
        {
            oldPos = Particle[i].pos;
            Particle[i].pos = oldPos + VelocityArray[i];

            //Check to see if the particle goes beyond volumetric limits
            lengthParticle = Particle[i].pos - origin;
            if (lengthParticle > length)
                reinitialize(Particle[i]);
        }
    }
}
```

Figur e 35: Pseudocode for Particle System Algorithms

casualty causing condition is stopped and the ventilation fans are turned on. The dissipation of the obscurity is also simulated using a time dependent function.

E.   REAL-TIME CASUALTY SCENERIOS

Simulating a main space fire or an engine-room steam leak on board a Navy ship is both hard to coordinate and hard to simulate. These types of casualties are "all hands" events, affecting every person on board the ship. They must be done periodically to maintain the crew's proficiency and teamwork in fighting the casualties. Therefore, in order to conduct a fire drill or steam leak drill, it consumes many man-hours and requires a great deal of planning. Furthermore, the environmental effects is very difficult to simulate. Hence, scenarios incorporating the effects described above were integrated into the simulator to provide low cost, easy to coordinate, realistic training in addition to the periodic "all hands" drills. These casualties are capable of being networked to other workstations to allow the entire team to interact and coordinate combatting the casualty. This networked capability is discussed in detail in.

1.   Main Space Fire Casualty Sequence

By depressing either of the 'f' or 'F' keys on the keyboard, the main space fire casualty sequence commences with a JP-5 fuel oil leak at a piping elbow joint in the lower level of the engine room. The fuel oil leak develops into an engine room fire if the oil leak is not stopped within twenty seconds by shutting an isolation valve upstream of the leak. The fire breaks out with a radius of two meters and, if no extinguishing agent is applied, grows with each frame cycle until it reaches a maximum radius of 3.5 meters.

The fire can be extinguished by either obtaining and opening a vari-nozzle to apply high velocity spray to the base of the fire or activating the halon fire extinguishing system. If using the vari-nozzle to apply the extinguishing agent, the fire-fighter must be within six meters of the fire and apply the high velocity spray within five degrees of either side of the fire's origin. The fire decreases in radius at a rate commensurate with the amount of time the extinguishing agent is applied. If the firefighter does not keep the high velocity

spray within the above constraints, the fire will grow as before. If, instead, the firefighter activates the halon fire suppression system to extinguish the fire, the fire responds as in reality and decreases at a quicker rate than if water is applied.

Once the fire is initiated, the environment in the main space begins to fill up with smoke. Gray-black smoke incrementally fills the compartment for as long as the fire continues to burn, causing a reduction in visibility until a minimum visibility of five meters is reached. Once the fire is out, the smoke can be cleared by turning on ventilation fans.

2. Steam Leak Casualty

By depressing either of the 's' or 'S' keys on the keyboard, a steam leak develops at a union on the deaerating feed tank (DFT) outlet piping just below the DFT feed isolation valve. The steam leak is constrained within the bounding volume of a cone, with a length of 0.75 meters and a radius of 0.25 meters. The size of the steam leak changes if the DFT feed isolation valve is manipulated. As the valve handwheel is closed, the leak reduces in proportion to the percentage the valve is opened. This is consistent with reality, since as the valve is shut, it will throttle the flow of steam through it, reducing pressure at the leak. Once the DFT feed isolation valve is fully shut and ventilation fans are activated, the steam can be dissipated.

The steam leak also causes the atmosphere to become obscured as in the fire casualty discussed above. The difference is that the color of the obscurity for steam is white-gray, compared to the grey-black of the fire. If both casualties are occurring at the same time, the combination of the smoke and fire causes the obscurity to take on a mixed color, where the amount of color caused by each casualty depends on the length of time the particular casualty has been in affect.

F. SUMMARY

Simulating environmental conditions is very important in creating realistic and effective training scenarios. This simulation uses both Performer utilities and the authors' own code to create an environment significantly realistic to train sailors. The advantage of

the authors' code is that it can be easily manipulated to create several different types of casualties.

IX. NETWORKED ENVIRONMENT

A. RATIONALE FOR A NETWORKED TRAINER

Since the earliest days of naval warfare, the efficiency of a ship in battle has been largely determined by teamwork of the sailors manning her. A ship whose crew is trained to work together smartly will consistently defeat a ship whose teamwork is not as polished, even if the skills of the individual sailors on the losing ship exceed those of the winners. The Navy has long understood this truism, and place heavy emphasis on training shipboard personnel as a team. Traditionally, team training has either been carried out at sea or by sending a ship's team to a land-based team trainer at one of a limited number of locations. However, as training budgets decline, both operating funds to send ships to sea and training funds to send teams to remote training sites are declining to alarmingly low levels.

To maintain a high degree of readiness, a new, lower-cost training method is required. The reasons why virtual reality is an outstanding vehicle to meet this need are given in the introduction. However, unless the virtual environment is networked, it cannot meet the Navy's team training requirements, and therefore will not teach the most important lesson of training: teamwork.

B. THE DIS COMMUNICATIONS PROTOCOL

In the history of computers, some of the biggest problems have involved compatibility. Traditionally, all aspects of computer systems have been designed with little thought to interoperability and interacting with other systems. A multitude of operating systems, architectures, languages, and networking protocols exists, yet there is no standard for any of them to interact. Because of this, different systems cannot be used together, forcing the consumer to completely revamp his entire operation at considerable expense every time a new development in technology occurs.

In order to avoid this pitfall in the area of simulations, the government decided that all simulators built for its use must meet a compatibility standard which allows them to

communicate with each other. This standard is the distributed interactive simulation (DIS) protocol, and all simulators built for the government must be able to communicate using this system [DOD92]. Accordingly, the ship walkthrough project uses the DIS protocol to allow different workstations participating in the same exercise to communicate.

The original version of this protocol is described in detail in [IST91], and the version 2.0.3, which is used by this simulation, is contained in [IST93]. The basic concept of this protocol is to decentralize the database, so that each individual participant keeps its own copy of the database. This allows participants using a wide variety of systems to take part in the same exercise. For example, an M-1 tank simulated on a $350,000 SIMNET node at Fort Knox, Kentucky, can be fighting a Hind helicopter simulated on a $25,000 Indigo 2 Extreme at the Naval Postgraduate School in Monterey, California.

1.   Protocol Data Units and Deduced Reckoning

Exercises begin with each entity (participant in the simulation) sharing the same terrain database. Entities communicate by sending protocol data units (PDU's) across the network. Each PDU contains a unique exercise identification, so that several exercises can share the same network without interference. There are several types of PDU's which can be used to describe various events, such as weapons firings and detonations, logistics requests and events, and collisions. However, the basic type of PDU which carries a majority of the networking traffic is the entity state PDU. The structure of the entity state PDU is described in detail in Table 3.

The entity state PDU is used by an entity to give other participants essential information about itself. It contains data fields which allow the entity to encode a wide range of information, which include its unique identification, force identification, location, linear velocity, acceleration, and markings.

It is possible for a participant to send out entity state PDU's every frame to constantly inform all the other participants of its actual location, velocity, etc. However, with a large number of participants, this would quickly overwhelm the capability of the

Field Size (Bits)	PDU Fields	PDU Subfields
96	PDU Header	Protocol Version - 8-bit enum.
		Exercise ID - 8-bit unsigned int.
		PDU Type - 8-bit enumeration
		Padding- 8-bit unused
		Time Stamp - 32-bit unsigned int.
		Length - 16-bit unsigned integer
		Padding - 16-bit unused
48	Entity ID	Site - 16-bit unsigned integer
		Application - 16-bit unsigned int
		Entity - 8-bit unsigned integer
8	Force ID	8-bit enumeration
8	Artic'ted Parameters	8-bit unsigned integer
64	Entity Type	Entity Kind - 8-bit enumeration
		Domain- 8-bit enumeration
		Country - 8-bit enumeration
		Category - 8-bit enumeration
		Subcategory - 8-bit enumeration
		Specif ic - 8-bit enumeration
		Extra - 8-bit enumeration
64	Alternate Entity Type	Entity Kind - 8-bit enumeration
		Domain - 8-bit enumeration
		Country - 8-bit enumeration
		Category - 8-bit enumeration
		Subcategory - 8-bit enumeration
		Specif ic - 8-bit enumeration
		Extra - 8-bit enumeration

Table 3: Entity State PDU. From [IST93]

Field Size (Bits)	PDU Fields	PDU Subfields
96	Entity Linear Velocity	X - Component - 32-bit FP
		Y - Component - 32-bit FP
		Z - Component - 32-bit FP
192	Entity Location	X - Component - 64-bit FP
		Y - Component - 64-bit FP
		Z - Component - 64-bit FP
96	Entity Orienta-tion	Psi - 32-bit floating point
		Theta - 32-bit floating point
		Psi - 32-bit floating point
32	Entity Appearance	32-bit record of enumerations
320	Dead Reckoning Parameters	Dead Reckon Algorithm - 8-bit enum
		Other Parameters - 120 bits unused
		Entity Linear Acc. - 3x32-bit FP
		Entity Ang. Vel. -3x32 bit Integer
96	Entity Markings	Character set - 8-bit enumeration
		11 - 8-bit unsigned integers
32	Capabili-ties	32 boolean fields
n x 128	Articula-tion Parameters	Change - 16-bit unsigned integer
		ID-attached to-16-bit unsigned int
		Parameter Type - 32-bit parameter type record
		Parameter value - 64 bit

Table 3: Entity State PDU. From [IST93]

network and bring the simulation to a halt. Instead, a participant sends out an entity state

PDU as rarely as possible. In the time lapse between receiving the PDU's of another entity,

an entity in the exercise estimates the position of the other entity using deduced reckoning

(dead reckoning, or DR). This process is the reason the entity state PDU contains fields for linear velocity, acceleration, and dead reckoning parameters. Dead reckoning takes an entity's last known position (the location field of the last entity state PDU received) and, using the velocity and acceleration of that PDU and the time since that PDU was received, estimates that entity's current position. The simulation then places the graphical representation of the entity at that location in the virtual world.

An obvious problem with this method is that an entity may maneuver in the time between sending PDU's. This causes its actual position to differ significantly from the DR'ed position where all the other entities in the exercise are representing it. This is solved by having each entity DR itself and compare the DR result to its actual location. If the difference between the two is greater than a predetermined threshold value (normally three meters), the entity will send out another entity state PDU to inform all the other participants of its correct location. If the time since its last PDU exceeds a certain value, an entity will send an entity state PDU, even if its DR'ed position still matches its actual position. This is done to inform the other participants that it is still participating in the simulation. This time is five seconds unless another value is specified before the start of the exercise [IST93].

C. NETWORKING THE SHIP WALKTHROUGH

1. Networking Other Participants

a. *Method of Communicating User's Information*

The first step in creating a shared virtual environment is to create a method of communication so that each workstation involved can update the others about the actions of its entities. This is fairly simple to do by using the DIS networked library developed by Zeswitz for his master's thesis here at NPS [ZESW93]. This library allowed the programmers to merely create PDU's and use the library function "write_pdu" to transmit them over the network. Receiving PDU's was equally simple, using the "read_pdu"

function. In addition, much of the information contained in the entity state PDU, such as entity type, appearance, markings, etc., is ignored, since it is not required to convey any information by this application. Also simplifying matters is the fact that the simulation was originally limited to under twenty participants, a fairly small number by DIS standards. This allows a simple matrix to be used to store the applicable information of each entity in the simulation instead of a complex hashing algorithm.

As mentioned above, an entity in the exercise must determine when to send a PDU over the network to update the other users. As discussed above, PDU's are sent if the time since the last PDU sent exceeds a certain value, or if the entity's position and orientation differ from its DR'ed position and orientation by greater than a certain amount. This is a drawback to networking a human figure instead of vehicles, which are more prevalent in networked environments. Because humans can change their orientation much quicker than a truck or a tank, it results in a much larger network load. A human can turn ninety degrees in under one fifth of a second, while a tank might take as long as fifteen seconds. This means that humans are much more likely to exceed the allowable difference between actual orientation and DR'ed orientation, necessitating that a PDU be sent. An example of this is that a simulation in which most of the entities are vehicles, such as NPSNET, has an average PDU rate of eight PDU's / entity-sec depending on the type of exercise, whereas the ship walkthrough's average is fifteen PDU's / entity-sec

Another problem concerning sending PDU updates is the ability of the user to stop and start instantly. Because acceleration, and more important, deceleration, are quick, discrete processes vice the slow analog processes they are in a vehicle, the most likely source of error between an entity's actual position and its DR'ed position is a change in speed of the user. The jump in moving an entity's representation from its DR'ed position to its updated position is quite noticeable. Therefore, whenever the user stops, a PDU is immediately sent informing all the other participants of the entity's location and velocity.

Although the entire entity state PDU must be transferred over the network, only a few fields are actually needed to convey the information required by this simulation:

the entity id, which uniquely identifies each entity in the simulation; the force_id field, which is used to identify entities is exiting the exercise; the entity location, which gives that entity's location in three space; the entity orientation, which gives the entity's direction of view in three space; and the entity's velocity, which gives the entity's direction of motion in three space. To take advantage of this, an entity state PDU is created when the network is initialized. The entity id of this particular user, which will never change during the exercise, is stored in this PDU and the unused portions of the PDU are set to null values. Whenever the unit needs to send a PDU, it simply updates the portions of this PDU that the application uses and sends it. Thus, all the unused fields of a PDU are only set once, reducing the overhead of sending a PDU.

b.   Handling PDU's

In order to save memory space, rather than storing entire PDU's for each entity, a data structure, named "sailorType", was created, which contains only the information necessary to the application. This structure is shown in Figure 36. The integer "empty" indicates if the structure currently contains an entity or is empty. The entity's location and heading information is stored in "DRposit," which is of type "pfCoord," a special Performer data structure. It velocity is contained in "DRvec," a three dimensional vector. The use of the Performer DCS node and Performer switch node will be discussed later in this chapter. A one-dimensional matrix of "sailorTypes", whose dimension is the maximum number of participants allowed, is created during the network initialization process and named "sailors."

```
// Sailor type structure
typedef struct
{
    int            empty;
    EntityID       entity_id;
    pfCoord         DRposit;
    pfDCS          *body;
    pfSwitch        *Switch;
    pfVec3          DRvel;
} sailorType;
```

Figure 36: Sailor Type Structure

Once an entity receives a PDU from another entity, it must interpret that PDU to correctly update that entity's position in the world. The function which performs this action is shown in Figure 37. It first checks if the PDU is from itself, and if so, it discards it since it has no need to update itself. If the PDU is not its own, it then loops through all the sailorType entities in its database, comparing the entity id of this unit to those of the units already in the database. If it finds a match, it checks if that entity is leaving the simulation. If so, it deletes it from the world; if not, it uses the information found in the PDU to update that entity's location, orientation, velocity and move its representation in the virtual world. If the entity is not already in the database, it checks if there is room for another entity in the database. If so, it adds the entity to the database; otherwise, it prints an error statement.

```
if (own PDU) exit function
while (i < number of entities in database) loop
{
            if (PDU.entity_id = entity(i).entity_id in database)
                    if (entity(i) is leaving exercise)
                            delete entity(i)'s model
                            exit function
                    else
                            update entity(i)'s location
                            translate entity(i)'s representation there
                    exit function
            i = i + 1;
}
end loop
            if (room for another sailor)
                    add entity to database
            else
                    print error statement
```

Figur e 37: Pseudo Code Describing the PDU Handling Function

c. *Representing the Other Entities in the Virtual World*

Up to this point, the discussion has been limited to sending and receiving PDU's. However, even if each work station in a virtual world knows where all the other entities are, that information is useless unless it is used to produce a graphical representation the user can see. In order to enhance the feeling of realism, these other

participants are represented using a primitive, non-articulated model of a human, displayed in Figure 38. The model was created here at NPS by Paul Barham based upon University of Pennsylvania's JACK model as a MultiGen FLT file for use as a solider in NPSNET. It is renamed "blueshirt.flt" and modified for this project to suggest a sailor; his torso and legs are colored light and dark blue, respectively, to indicate dungarees, a sailor's basic working uniform afloat, and he is wearing a ballcap, a sailor's at-sea headgear. This model, although far from lifelike, provides a recognizable image of a sailor, from which the user can determine the orientation, position, and movement of other participants in the networked environment.

During the model loading process, "blueshirt.flt" is loaded and stored in shared memory. Once a PDU has been received, if the sending entity is not already in the database, a copy of the model is created and stored in memory. The pointer to the DCS node for this entity in the matrix "sailors" is set to the copy's location in memory. This DCS node is then translated to that entity's position and aligned along its orientation. If the entity is already in the database, the existing DCS node is translated and aligned correctly. In the time between PDU's, the model is translated to its DR'ed position based upon its last reported location and velocity.

2.   Updating the Model of the Virtual Ship

The networking effort was expected to end once it was possible to connect different participants and visually represent each participant on all the workstations involved in the exercise. However, once was achieved, the result was unsatisfactory. What one user did to the database was not represented at the other workstations, and this effect severely reduced the realism of the simulation. For example, a user would open a door in his world and walk through it. However, since the other stations were not informed that the door opened, it appeared to other users that the sailor magically walked through a closed door. While this is a humorous example, deficiencies such as this severely reduce the realism of the environment. To create an effective training tool, this flaw had to be

Figur e 38: Graphical Repr   esentation of A Sailor

corrected. However, updating the underlying database is a much more complicated process than updating users' positions and movement. It is necessary to account for the race conditions which are created when more than one participant attempts to manipulate the same resource simultaneously. To make matters worse, there is not an explicit method to send database updates between participants in an exercise using the DIS standard.

Several methods to share updates between stations were considered. The final network code encodes the database information in the unused articulated parameters portions of the entity state PDU. The articulated parameter fields are normally used to give information about the entity whose position the entity state PDU updates, such as what direction a tank's turret is facing or whether a submarine's periscope is raised. Using them to describe the database itself is a relatively unused concept, yet it makes sense for a walkthrough such as this. Since the movement of all movable items in the database is constrained, it is possible to give satisfactory descriptions in the small space available in an articulated parameter record. The main advantage of using this method is that it does not create appreciably more net traffic, which is important, since, as discussed earlier, the network load is already quite high. The data is transferred as part of a PDU which is already being sent to update the entity's position. The only additional network traffic created is that a small number of changes to the database require a PDU to be sent immediately to update the other participants database.

Updating the database information in this manner is accomplished by always sending information about a certain object in the same location of the articulated parameters matrix in the entity state PDU. During the loading process, each item in the database which can be updated is given a unique index. When an item is moved and requires its position update to be sent to the other participants, its index and current position are sent to the networking portion of the program. The network code then places this information into the correct location in the articulation parameter matrix of the next PDU to be sent. If, before that PDU is sent, that object is changed again, the networking code overwrites the old value with the new. So long as all participants are using the same database, the index for each movable object will be the same for each participant.

This method of database updating ignores race conditions, which were not found to be a problem in this relatively simple database. Since all movable items are constrained, such as doors and drawers, two participants attempting to manipulate the same object resulted in the same result which would occur in reality: the item would jerk back and forth

as one then the other would gain control, and whoever pulled or pushed last controlled the position of the item. Although this is not a very elegant solution, and would not work for items which were free to move anywhere, such as a toolbox, it serves more than adequately for this database.

3.   Updating Other Information Between Participants

Although generating a networked environment where all the participants share the same database greatly increases the realism of the simulation, it still leaves areas which are unsatisfactory. The major problem is that the casualties are still local to the workstation which generates them. For example, if one participant starts a fire in the engine room on his station, the other participants in the engine room see only him move around as he carries out the actions to fight the fire. However, none of the others see the fire he created, the smoke from that fire, or the fact that he is holding the vari-nozzle to extinguish the flames. Therefore, networked team training could not be accomplished using this simulation as it then existed.

a.   *Updating the Representation of the Sailor*

The sailor can be in one of three states during the simulation; he can be not holding a vari-nozzle, he can be holding a vari-nozzle which is not discharging agent, and he can be holding a vari-nozzle which is discharging agent. To be able to display the distinction, the model displayed in Figure 38 was modified using MultiGen. The basic representation was copied twice, and one of these copies was modified to reflect a sailor holding a shut vari-nozzle, which is shown in Figure 39, while the other was modified to show a sailor holding an open vari-nozzle, which is shown in Figure 40. The original and two copies were placed in the hierarchy of "blueshirt.flt "as children of a switch node. A switch node is a node which can display either zero, one, or all of its children. A pointer to this entity's switch node was placed in the correct index of the matrix "sailors". Originally, all the participants are shown as the sailor without the nozzle. If a participant picks up a nozzle, the next PDU he sends reflects that fact. When each of the other participants in the

exercise receive that PDU, they change which of the children of the switch node is being displayed to the sailor with the shut nozzle. Likewise, if a participant then opens the nozzle or returns it to storage, he sends a PDU which informs the others of that fact and they display the correct representation of that sailor.
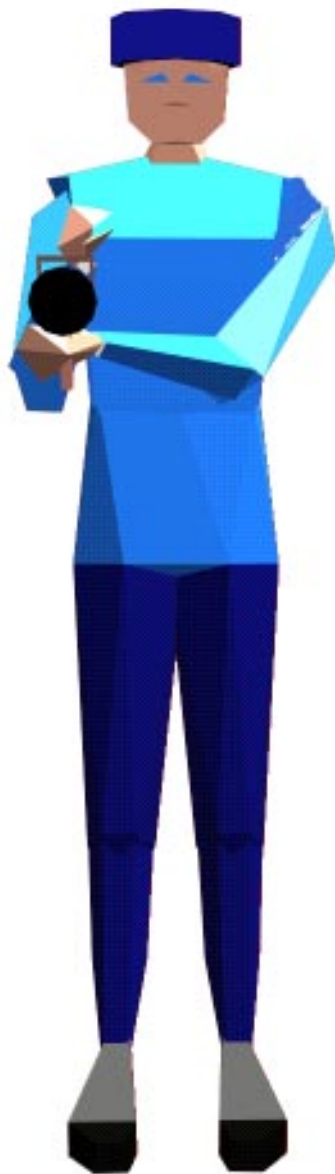
Figur e 39: Sailor Holding a Closed V ari-Nozzle

Figure 40: Sailor Holding an Open Vari-Nozzle

b. Updating Casualties

The solution which was used to solve the problem of networking the database had to be modified before it could be applied to updating casualties, because in

this case race conditions can not be ignored. For example, a fire should increase in scope if no action is taken and decrease in scope if a participant puts extinguishing agent on it. However, if one participant takes no action to extinguish the fire and sends out the size of the fire being larger than before, and another sprays water on the fire and sends out the size of the fire being smaller than before, the fire's behavior is nondeterministic: it could shrink as it should, it could grow, or it could stay the same. Obviously, this is not an acceptable solution.

To correct this, a form of "ownership" is used for the casualties in the simulation, (main space fire, fuel oil leak and steam leak), and their corrective actions (applying extinguishing agent and shutting valves). The distributed nature of the network is maintained because no one central workstation is responsible for all casualties. Instead, the station which initiates the casualty "owns" it, i.e., is responsible for updating the other stations as to the status of the casualty. For example, if station "A" starts a fire, it keeps track of the size of the fire. (The terms used to describe the casualties and their parameters are described fully in the Chapter VIII.) This means that if station "B" sprays the fire with water, instead of "B" reducing the size of the fire and sending out an update of the new size, the next PDU it sends contains the information that it is putting water on the fire. All the other stations ignore this portion of the PDU, but station "A" takes it, reduces the size of the fire, and the next update PDU it sends will reflect the reduction in fire size. If more than one station is putting water on the fire, each will tell station "A", which will decrement the fire size based upon how many stations are extinguishing the fire. This is realistic, since if several hoses are used on a fire, it will decrease quicker than if a single hose is used.

Updating the steam and fuel oil leaks is done similarly. The participant which started the leak keeps track of the position of the regulating valve. As another participant manipulates the valve, he sends a PDU saying he is opening or closing the valve. The station which initiated the casualty takes this message and updates the valve position. The initiating station then sends out a PDU informing all the other participants of the valve's position, so they can display their representation of the leak correctly.

D. SUMMARY

As work on this thesis began, networking the application was not considered to be in the scope of this thesis. However, as the application progressed, it became obvious that a simulation which does not allow team training would be of little or no use to the Navy, so a networked system was created.

This networking solutions used for this application work very well for its scope and complexity. However, the methods were devised more as a proof of concept than a final solution to the problem. The network was created to show that it was feasible to network several sailors and casualties in the same virtual shipboard environment. As the simulation grows in complexity, it is unlikely that the method used will be able to expand sufficiently to handle a broader range of casualties. This is because of the size of the PDU's effects network efficiency. While it is theoretically possible under [IST93] to add any number of articulated parameters to an entity state PDU, in practice this would make the entity state PDU so large that network efficiency would be adversely effected. There are several methods which can be used to overcome this problem, and these are discussed in the future work section of Chapter X.

X. CONCLUSION

A. RESULTS

This goal of this thesis is to create a prototype shipboard virtual environment, which can prove the feasibility of using virtual environments as a training tool for the U.S. Navy. To achieve this goal, this thesis explores several areas of computer science, and achieved the following results:

- A model of sufficient size and complexity was created to demonstrate that a large-scale database can be visualized at real-time, interactive frame rates. To do this, the model is stored in a hierarchical data structure and divided into potentially visible sets, which greatly increases the performance of the algorithm.

- The simulation is networked so that several user's can interact in the same virtual environment for team training. Not only can the users see and interact with each other, the database is also networked to increase the realism of the simulation

- An involved collision detection mechanism was created to facilitate picking and prevent the user from moving through decks, bulkheads and other objects.

- Several training scenarios were created to both test and train the user. The scenarios realistically respond to the user's actions to give effective training feedback.

- A wide range of environmental effects were created which simulate casualties to increase the realism of training scenarios.

- An HMD version of the walkthrough was created to give the user a sense of immersion.

- A desktop version was created to allow the user to spend several hours in the virtual environment.

- Several training devices are incorporated into the simulation. These include a hypertext window to display information, path planning to teach the user navigation skills, and a deck overview to constantly inform the user of his position in the ship.

The literature search did not reveal any walkthrough system which combined such a wide range of features. The method in which these features were combined creates a highly realistic, easy to use simulation with a great deal of potential as a training platform. This program could easily be expanded to become a extremely valuable training device for the Navy of the Twenty-First Century.

B. RECOMMENDATIONS FOR FUTURE WORK

There are several areas which need to be implemented or improved before this project is ready to be used as a fleet trainer. They are included below, in the authors' order of importance.

1. Create a Model Using Real Ship Data

As impressive and functional as the result of this thesis is, its use will be limited until the virtual environment is simulating an actual Naval vessel. Since an actual Navy ship is too large to model without using CAD data, a method to convert the CAD data used into a format which can be visualized is a necessity. However, as long as contractors use their own proprietary CAD software to design ships, building a visualization model will be so cost prohibitive to be almost impossible. The Navy needs to create a standard format for visualization data and require that all CAD data delivered by contractors be in that format. This would be much like the DIS protocol imposed on all simulators. Although there would be quite an outcry from the contractors about this requirement, in reality it is not overly onerous and would end up saving the government a great deal of money. The day when everything will be visualized before it is built is almost here; the Navy needs to plan for it.

2. Articulated Human

Currently in the networked version of the shipboard VET, entities are represented as non-articulated humans. The only articulation is having three versions of the model of the entity, each of which indicates a different status of the vari-nozzle. As an entity moves through the ship, it appears to "float" over the deck, with no motion of its arms or legs. This is very unrealistic, and greatly reduces the user's feeling of immersion.

The NPS graphics group has already integrated University of Pennsylvania's articulated human, JACK, into NPSNET. Inserting JACK into the shipboard VET is a very straightforward matter, and will add greatly to the realism of the application.

3.   Improving the Networked Capability

The current networking system handles the database and required number of participants quite well. However, the method used to transmit the database information cannot be expanded to a significantly larger database as is required if this system is to model an entire ship. There are several methods which will solve this problem; using another type of PDU besides the entity state PDU or creating several entities for each database to increase the amount of articulated parameter slots available are potential solutions. Research needs to be done to determine the best method to update a large-scale database with a high number of networked users.

4.   Better Interfaces and Input Devices

To take maximum advantage of the HMD version of the simulation, better interfaces must be implemented. The current configuration of the NPS Graphics Lab HMD makes it difficult for the user to examine objects which are "behind" his initial orientation; this is most noticeable when the user attempts to reverse his path. The proposed changes in the lab configuration will help correct this problem.

In addition, it is difficult for the user to interact with a three dimensional world using the mouse, a two dimensional input device. If a three dimensional input device, such as a 3-D mouse or data glove, were used, it would allow the user to pick objects in a more realistic manner, greatly improving the realism of the simulation. It would also give the user the capability to travel in a direction other than his view direction, which is a current limitation.

The third method to improve the user's sense of realism is to incorporate the SARCOS I-PORT device. This device is a stationary unicycle that converts the user's pedaling into translation in a virtual world. It allows a user in an HMD to pedal through the world; this would greatly increase the realism of the simulation by making leg motion required to move through the world. The I-PORT has force feedback pedals, which

increases the pedal resistance to make the user pedal harder as he climbs ladders. Also, this device will allow the user to move in a direction other than his view direction.

5.   Varied Casualty Scenarios

There are only three casualties which can currently be simulated in the shipboard VET. In addition, these three casualties can only occur at fixed locations in the ship, and the only variables in the entire scenario are the user's actions. Having such limited scenarios make it easy for the crew to spot only one or two indications and know exactly what the drill is and how to fight it optimally. Reality, however, is never so simple, and training such as this actually makes the team less able to react to situations other than the canned scenario. Therefore, the ability to create many more different types of casualties, and the ability to randomize the location, scope, and scripting of the casualties is essential to have an effective trainer.

6.   Semi-Autonomous Forces

Given the hectic schedule of sailors at sea, finding a time when a large group can get together to use the simulator is difficult. A single sailor can currently use the simulator, but he will get no form of team training. However, if he could interact with semi-autonomous sailors, guided by a form of artificial intelligence, who would simulate the other members of his team, the training value would be greatly increased. Therefore, a form of semi-autonomous "teammates" needs to be created to allow individual sailors to get training as part of a team.

7.   Increased Data Display

Currently, when the user selects an object, only the name and function of the object are displayed. This capability should be expanded to display a wide range of information, such as what system the object is in, what is its normal position, what effect manipulating it will have on the ship, etc. Also, in the case of systems, it should be possible

to call up and display the diagram of the system for the user to immediately learn about how each object fits into the larger picture.

8.   Improved PVS Algorithm

As the proposed improvements in the model are implemented, the polygon count will increase exponentially. In order to maintain a satisfactory frame rate, the PVS algorithm will have to be improved. The manual method of defining cells and PVS's will no longer be adequate; an algorithm to compute smaller visibility cells and more precise PVS's will be essential. In addition, the PVS algorithm needs to be expanded to include swapping textures as well as geometry.

9.   Testing and Evaluation

The virtual ship was created to train sailors, not merely be an exercise in computer science. In order to verify that VE is an effective method to train sailors, the effectiveness of the VET will have to be measured. While this is not a computer science task per se, it is essential that various methods of virtual training be evaluated for effectiveness in order that the computer scientists can create the best trainer possible. The Operations Research and System Management Departments of the Naval Postgraduate School should create experiments which can measure the training efficiency of the simulation.

10.   More Realistic and Efficient Collision Detection

The current method of using line segments for collision detection should be upgraded to a more efficient volume intersection algorithm. This would reduce the amount of overhead involved in the simulation. Also, the algorithm should be changed so that when the user hits a wall, he doesn't stop, but instead bounces off at the angle of reflection. This would allow the user to navigate tight spaces easier and simulate sailors returning from liberty walking down the passageway.

11.   Improved Path Planning

The current path planning algorithm uses relatively low level artificial intelligence. Although most of the time it works fine, there are locations where is makes the user transit away from where he wants to go to reach the first checkpoint. Also, if additional intelligence is added to the path planning algorithm, it can take into account that an area may be impossible to pass through due to battle fire, smoke, flooding, battle damage, etc., and route the user along a path that can actually be followed given the actual conditions.

Another possible way to use the path planner would be to make a handled version, which the use can carry with him through the ship. This would allow him to watch the path being displayed as he actually transits the ship.

LIST OF REFERENCES

[AIRE90A] Airey, John M., Rohlf, John H. and Brooks, Frederick P. Jr. "Towards Image Realism with Interactive Update Rates in Complex Virtual Building Environments," Computer Graphics, Vol. 24, No. 2, March 1990, pp. 41.

[AIRE90B] Airey, John Milligan "Increasing Update Rates in the Building Walkthrough System with Automatic Model-Space Subdivision and Potentially Visible Set Calculations," UNC Technical Report TR90-027, July 1990 (Airey's Ph.D. Thesis).

[ALSP92] Alspaugh, John C., "A Short Guide to AutoCAD Drawing Primitives for 3D Computer Graphics Models and the Walkthrough AutoCad-to-Polygon Conversion Program", UNC Technical Report, April 1992.

[BLAN92] Blanchard, Chuck, and Lasko-Harvill, "Humans: The Big Problem in VR," SIGGRAPH 92 Course Notes, Course 9 "Implementation of Immersive Virtual Environments," Chicago, July 1992.

[BROO86] Brooks, Frederick P. Jr. "Walkthrough - A Dynamic Graphics System for Simulating Virtual Buildings," Proceedings of the 1986 Workshop on Interactive 3D Graphics, Chapel Hill, 23 - 24 October 1986, pp. 9-21.

[BROO92] Brooks, Frederick P. Jr. "Final Technical Report - Walkthrough Project June, 1992 to Computer and Information Science and Engineering National Science Foundation," UNC Technical Report 92-026, June 1992.

[CATM84] Catmull, Ed, Carpenter, Loren, and Cook, Rob, Private and public communications, 1984.

[CORB93] Corbin, Daniel Patrick, "NPSNET: Environmental Effects for a Real-Time Virtual World Battlefield Simulator," Master's Thesis, Naval Postgraduate School, Monterey, California, September 1993.

[CLAR76] Clark, James H. "Hierarchical Geometric Models for Visible Surface Algorithms," Communications of the ACM, Vol. 19, No. 10, October 1976, pp. 547.

[DOD92] Department of Defense, "Defense Modeling and Simulation Initiative," Washington D.C., May, 1992.

[FUNK92] Funkhouser, Thomas A., Sequin, Carlo H and Teller, Seth "Management of Large Amounts of Data in Interactive Building Walkthroughs," Computer Graphics, Proceedings of the 1992 Symposium on Interactive 3D Graphics, March 1992, pp. 11.

[FUNK94]   Funkhouser, Thomas A., Khorrambadi, Delnaz, Sequin, Carlo H and Teller, Seth J. "UCB System of Interactive Visualization of Large Architectural Models," April 1994.

[GARD85]   Gardner, Geoffrey, Y., "Visual Simulation of Clouds," Computer Graphics Proceeding, Vol. 19, No. 3, 1985, pp. 297-303.

[GARD92]   Gardner, Geoffrey, Y., "Battlefield Obscurants Final Technical Report," Grumman Data Systems Corporation, Woodbury, New York, September 1992, pp. 8-31.

[GVU95]   Graphics, Visualization and Usability Center Virtual Environments Group Home Page, "Managing Level of Detail with a High Detail Inset," http://www/cc/gatech.edu/gvu/virtual/Detail_Inset/, Georgia Institute of Technology, 1995.

[HITL94]   Human Interface Technology Laboratory, Washington Technology Center, University of Washington, "A Crisis Management Testbed for Experimental Training of Damage Control Assistants," March 31, 1994.

[HUBB93]   Hubbard, Philip, "Interactive Collision Detection", "Proceedings of IEEE Symposium on Research Frontiers in Virtual Reality", October1993.

[IST91]   Institute for Simulation and Training, "Protocol Data Units for Entity Information and Entity Interaction in a Distributed Interactive Simulation (DRAFT)," Version 1.0, IST-PD-90-2, Orlando, FL, September 1991.

[IST92]   Institute For Simulation and Training, "Distributed Interactive Simulation: Operational Concept," Draft 2.1, Orlando, FL, September 1992.

[IST93]   Institute for Simulation and Training, "Standard for Information Technology- Protocols for Distributed Interactive Simulations," Version 2.0, Third Draft, IST-CR-93-15, Orlando, FL, May 28, 1993.

[JALI94]   Jalili, Reza, Kirchner, Peter D., et al., "A Fly-Through of the Frauenkirche," April 29, 1994.

[LEVI92]   Levit, Creon and Bryson, Steve, "Lessons Learned While Implementing the Virtual Windtunnel Project," SIGGRAPH 92 Course Notes, Course 9 "Implementation of Immersive Virtual Environments," Chicago, July 1992.

[LIN93]   Lin, Ming C., "Efficient Collision Detection for Animation and Robotics", Ph.D. thesis, University of California, Berkeley, December, 1993.

[LOCK]   Lock, John, Pratt, David R., and Zyda, Michael J., "A DIS Network Library for UNIX and NPSNET," Naval Postgraduate School, Monterey, California, undated.

[MINE94]   Mine, Mark R., and Weber, Hans, "Mega-Models: Breaking the Back of Virtual Environments Research," University of North Carolina Technical Report, unnumbered, 1994.

[NATI94]   National Academy of Sciences National Research Council Committee on Virtual Reality Research and Development, "Report on the State-of-the-Art in Computer Technology for the Generation of Virtual Environments" January 1994.

[PRAT92]   Pratt, David R. et. al., "NPSNET: A Networked Vehicle Simulation With Hierarchical Data Structures," Proceedings of IMAGE VI Conference, Scotsdale, AZ, July 1992.

[PRAT93]   Pratt, David R., "A Software Architecture for the Construction and Management of Real Time Virtual Worlds," Dissertation, Naval Postgraduate School, Monterey California, June 1993.

[REEV83]   Reeves, William T., "Particle Systems - A Technique for Modeling a Class of Fuzzy Objects," ACM Transaction On Graphics, Vol. 2, No. 2, April 1983, pp. 359-376.

[ROBI92]   Robinett, Warren and Holloway, Richard "Implementation of Flying, Scaling and Grabbing in Virtual Worlds," Computer Graphics, 1992 Symposium on Interactive 3D Graphics, March 1992, pp.189.

[ROHL94]   Rohlf, John and Helman, James, "IRIS Performer: A High Performance Multiprocessing Toolkit for Real-Time 3D Graphics", 1994 SIGRAPH Course Notes, May 1994.

[RUBI80]   Rubin, Steven M., and Whitted, Turner, "A 3-Dimensional Representation for Fast Rendering of Complex Scenes," ACM 1980.

[SGI94]   Silicon Graphics, Inc., Document Number 007-1680-020, "IRIS Performer Programming Guide," Mountain View, California, 1994.

[SSI94A]   Software Systems Incorporated, "Mutigen Modeler's Guide," Revision 14.0, San Jose, CA, March, 1994.

[SSI94B]   Software Systems Incorporated, "Mutigen Version 14.1 Release Notes," San Jose, CA, September, 1994.

[TELL91]   Teller, Seth J., and Sequin, Carlo H, "Visibility Preprocessing for Interactive Walkthroughs," Computer Graphics, Volume 25, Number 4, July 1991, pp. 61.

[WATT94]   Watt, Anne E., "Modeling of Real-Time Dynamic Effects," Master's Thesis, Naval Postgraduate School, Monterey, California, September, 1994.

[WEAV94] Weaver, Janet L., Perrin, Bruce M., Zeltzer, David, "Damage Control Training in a Virtual Environment,", Draft of Contract Summary Report, December 29, 1994.

[ZELT92] Zeltzer, David, "Autonomy, Interaction and Presence", Presence, Vol. 1, No. 1, pp 127, 1992.

[ZESW93] Zeskwitz, Steven R., "NPSNET: Integration of Distributed Interactive Simulation (DIS) Protocol for Communication Architecture and Information Exchange," Master's Thesis, Naval Postgraduate School, Monterey, California, September, 1993.

[ZYDA94] Zyda, Michael J., "CS-4473 Virtual Worlds and Simulation Systems," Course Notes, Naval Postgraduate School, Monterey, California, January, 1994.

APPENDIX A.   USER'S GUIDE

This appendix is the user's guide for operating the shipboard virtual environment trainer (VET). It covers starting and running the system and discusses the interface options and various commands available to interact with the virtual ship.

A.   STARTING SHIPBOARD VET

The Shipboard VET can be run on a variety of graphics platforms. During the initialization process, it determines the number of processors available on the platform and configures the multi-processing mode of the application.

To start the Shipboard VET, one must be in the directory in which the executable "walk" file is located. The executable is located in "/n/bossie/work3/king/ship/combined" directory at the Naval Postgraduate School Graphics Laboratory. By simply typing "walk" followed by a return, the program begins execution of a non-networked, standard monitor display shipboard VET.

To network the VET or direct the visual output to a head-mounted display, command line options are used following the walk command. To join an exercise in progress with other workstations, the -n command line option is required. If directing the visual output to a head-mounted display, the -h command line option is required.

The program takes approximately two minutes to complete the initialization phase. During the first portion of this period, the models and textures used for the simulation are loaded. Once loaded, a title screen consisting of the title of the project and its author's is displayed on the screen until the application is finished initializing (approximately twenty seconds). Following application initialization, the textures loaded earlier are downloaded into random access memory in order that they can be quickly accessed when needed. The textures are displayed on the screen as they are being downloaded. Once the texture download is completed the application begins and places the user in Combat Information Center.

B. PROGRAM TERMINATION

There are two methods to exit the Shipboard VET. One method is to press {Esc}, or your shell interrupt key, typically {Cntrl-C}. The other method is to select the quit menu button on the graphical user interface (GUI). Both of these options completely shuts down the system including any processes spawned during the application. (Note: the GUI option is not available when wearing the HMD)

C. SCREEN LAYOUT

The standard screen layout takes up the entire screen and includes the virtual scene display, graphical user interface (GUI) and deck overview. A pop-up window, which displays information about objects in the virtual ship, is displayed when objects are selected with the mouse. These displays and there relative locations on the screen are shown in Figure A-1 and Figure A-2.

The virtual scene display takes up ninety percent of the screen. The overview display and GUI can be turned off to allow the full screen to be taken up by the virtual scene display by either depressing 'F1' on the keyboard or selecting "GUI off" on the GUI. To reenable the GUI and deck overview display, 'F1' must be depressed on the keyboard.

1. Deck Overview

The deck overview channel is located on the lower right hand portion of the screen as shown in Figure A-2. It provides an overhead view of the deck on which the user is presently. The deck lay-out is graphically displayed in two dimensions showing the locations of ladders, bulkheads, doorways and passageways. A black position cursor shows the user's position in the virtual ship and moves as the user moves along the deck in the virtual environment.

Figure A-1: Screen Display

123
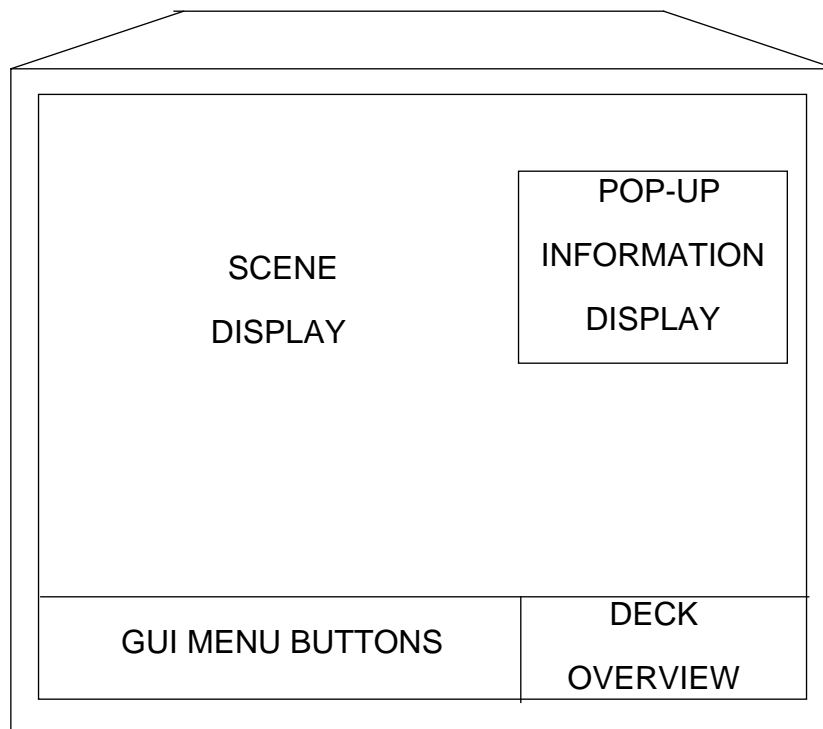
Figur e A-41: Monitor Display

2.   Pop-Up Data Display Window

When the user selects an object with the mouse, a pop-up window containing information about the object selected is displayed in the upper right hand corner of the screen as shown in Figure A-2. The display stays on the screen until the mouse buttons are released.

3.   Graphical User Interface

The graphical user interface (GUI) provides the user with an "easy to use" menu interface to perform an assortment of functions. The GUI is located on the lower left corner of the screen as shown in Figure A-2. A representation of the GUI is displayed in Figure A-3.

D. OPERATION

There are two modes of operating the shipboard VET. The first mode, "walk"mode which is the default mode, simulates naturally walking through the virtual ship. Collision detection is enabled meaning that the user cannot walk through objects. The other mode is "fly" mode which enables the user to move through the ship as if one were flying. In "fly" mode, collision detection is disabled allowing the user to fly through objects. These modes are changeable by the "Mode" menu toggle button on the GUI.

1.   Mouse Operations

Natural walking (walk mode) or flying (fly mode) is simulated with the aid of a mouse. By depressing either the right mouse key (forward motion) or the left mouse key (reverse motion), the user gains speed and translates through the environment in the direction the user is looking. The middle mouse button causes the viewer to stop.

The direction one is looking is also determined by the mouse. The view direction changes in the relative direction that the mouse cursor is positioned from the center of the screen. For example, the farther to the right of center the mouse cursor is, the quicker the individual will turn to his right. The range of motion in the vertical direction is capped to straight up (+90 degrees) and straight down (-90 degrees) when in "walk" mode. There is a one inch box in the middle of the screen referred to as the "dead zone" in which the mouse cursor, if inside this area, does not cause the view direction to change.

The mouse is also used to select objects (pick) in the virtual ship for either object data display, manipulation or movement. To select an object, the user places the mouse cursor on an object and depresses the middle and either the left or right mouse button at the same time. If the object is not a movable object, a pop-up window is displayed in the upper right hand corner of the screen as shown in Figure A-2 for as long as the mouse buttons are pressed down.

a. Objects Which Move

All doors throughout the ship and cabinet covers in the Radar Room can be opened and closed. To open a door, the right and middle mouse button must be depressed at the same time with the mouse pointing to the door. The door rotates in its open direction until it reaches its maximum rotation of ninety degrees or until the mouse buttons are released. To close the door, the left and middle mouse buttons are depressed at the same time, and the opposite motion occurs.

Two valves located in the Engine Room Lower Level are both capable of being opened and closed. The operation of valves is similar to the operation of doors as far as the method used to open and shut the valves. When opening a valve the valve stem rises and the valve hand-wheel rotates in the counter-clockwise direction; the opposite occurs when shutting.

A vari-nozzle, when picked, is moved from it's storage location in Engine Room Lower Level to directly in front of the user's view at belt level. The vari-nozzle can be opened and shut once the user has the nozzle in front of him by further picking of the nozzle. The nozzle is moved back to it's storage location by depressing 'p' or 'P' on the keyboard.

b. Objects Which Can Be Manipulated

A ventilation fan controller and halon activation system controller in Engine Room Lower Level are capable of being turned on and off by the mouse. To manipulate the controllers, the controller button must be picked as described above.

2. Graphical User Interface (GUI)

The GUI, displayed in Figure A-3, provides the following functions, starting in the upper right corner and proceeding clockwise:

QUIT	X Y Z H P R	WALK

GUI OFF	TRANSLATE TO CIC	OK

HEIGHT OF EYE

RESET	SHOW PATH TO OPS	OK
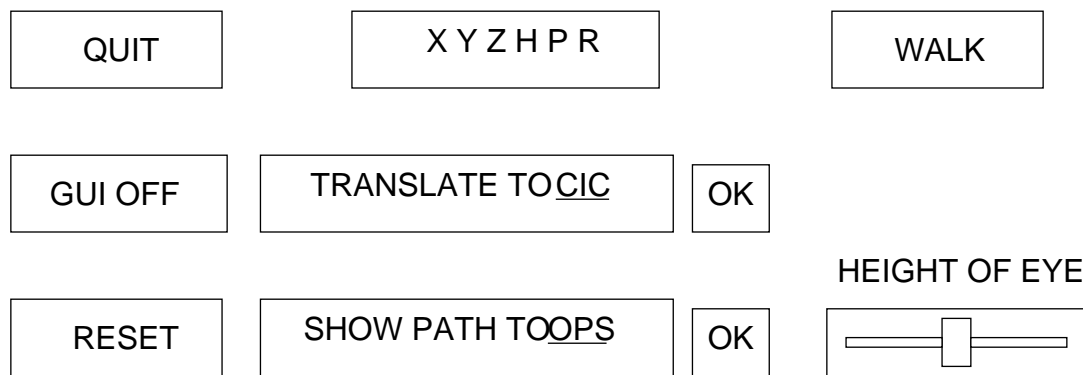
Figure A-42: Graphical User
Interface

a. Quit Button

Causes the user to leave the application. Pressing the ESC key also accomplishes the same function.

b. User's Position Display

The user's location in three space is displayed here as X, Y, and Z coordinates. Also, the user's heading, pitch and roll are also displayed.

c. Traversal Mode Selection

The two modes of operation, "fly" and "walk", which were previously discussed, are controlled by this toggle menu button.

d. Height of Eye Control

The height of eye control slider enables the user to vary the eye point height above the deck while in walk mode in order that objects which are close to the deck or up high could be viewed at a closer distance. The user can vary height between 0.5 - 2.5 meters.

e.   Path Planning Selection

A path planning tool is provided which takes the user along a path from his present location to a location of his choosing via the optimal route at normal walking speed. The locations which can be selected include CIC, the Radar Room, the Operations' Office, DCC, the Hull Technician's Shop and the ladder to the Engine Room. The user selects his destination by clicking the button titled "Show path to: <destination>." As he clicks it, a different destination is displayed. When the desired destination is displayed, the user selects the "OK" button next to it and begins travelling to that destination. At this point, the menu button changes to "Stop walking to: <destination>", and if the user selects it, he is no longer transiting to the destination and he regains control of his own motion.

f.   Reset Button

This button allows the user to reset the application to its original state. All objects are returned to their initial position, all casualties are terminated, any damage caused by casualties is repaired, and the atmosphere is cleared.

g.   Toggle GUI Button

The "GUI-Off" menu button turns the GUI and the deck overview off providing more screen display for the scene. The GUI and deck overview can be returned to the screen display by depressing "F1" on the keyboard.

h.   Translation Selection

To facilitate the user the ability to quickly "jump" from one location in the virtual ship to another in the virtual ship, a translate menu button is provided. Preset anchor points to key locations are embedded in the software code. These locations include Combat Information Center (CIC), Damage Control Central (DCC), Engine Room, Bridge and the Vehicle Loading Deck.

3.   Keyboard Operations

The keyboard is primarily used to initiate casualties on the user in the virtual environment. It also provides another method to accomplish some of the functions which are provided by the GUI. The keyboard inputs and their functions are listed in Table A-1.

Keyboard Input	Function
F1	Displays GUI and Deck Overview
ESC	Exits program
'd' or 'D'	Toggles CPU and graphics statistics
'f' or 'F'	Initiates fire casualty sequence
'p' or 'P'	Places fire nozzle back to stored position
's' or 'S'	Initiates steam leak casualty
't' or 'T'	Toggles texture display
'w' or 'W'	Toggles wireframe display
Shift 'b' or 'B'	Translate to Bridge
Shift 'c' or 'C'	Translate to CIC
Shift 'd' or 'D'	Translate to DCC
Shift 'e' or 'E'	Translate to Engine Room
Shift 'p' or 'P'	Translate to Vehicle Loading Platform
Print Screen	Saves RGB image of display on screen

Table A-1: Keyboard Inputs and Functions

4.   Head-mounted Display Operation

The program is configured to run with a head-mounted display (HMD) if the "-h" command line option discussed previously is used. The configuration changes the window size and graphics video format to be compatible with the HMD requirements.

"Walk" mode is the only mode of operation available when wearing an HMD. The GUI, deck overview and pop-up window are also not displayed with the HMD.

Walking through the virtual ship when wearing an HMD is very similar to the walking method discussed previously. The only difference lies in the method in which the view direction is determined when wearing an HMD. The HMD's tracking device translates the HMD's direction of view to an appropriate view direction in the virtual environment. Therefore, to walk around the virtual ship, the user physically looks in the desired direction and depresses the appropriate mouse buttons.

Movable objects such as doors, valves and the vari-nozzle and manipulated objects such as fan and halon controllers can still be picked while wearing an HMD. The method is very similar to the picking method discussed previously, however instead of selecting objects with the mouse cursor, select objects, by placing the cross-hairs in the center of the HMD view, on the object.

E. CASUALTY SCENARIOS

1. Fire Casualty Sequence

By depressing either of the 'f' or 'F' keys on the keyboard, the main space fire casualty sequence commences with a JP-5 fuel oil leak at a piping elbow joint in the lower level of the engine room. The fuel oil leak develops into an engine room fire if the oil leak is not stopped within twenty seconds by shutting an isolation valve upstream of the leak. The fire breaks out with a radius of two meters and, if no extinguishing agent is applied, grows with each frame cycle until it reaches a maximum radius of 3.5 meters.

The fire can be extinguished by either obtaining and opening a vari-nozzle to apply high velocity spray to the base of the fire or activating the halon fire extinguishing system. If using the vari-nozzle to apply the extinguishing agent, the fire-fighter must be within six meters of the fire and apply the high velocity spray within five degrees of either side of the fire's origin. The fire decreases in radius at a rate commensurate with the amount of time the extinguishing agent is applied. If networked and more than one individual is

putting out the fire, the fire goes down quicker. If the firefighter does not keep the high velocity spray within the above constraints, the fire will grow as before. If, instead, the firefighter activates the halon fire suppression system to extinguish the fire, the fire responds as in reality and decreases at a quicker rate than if water is applied.

Once the fire is initiated, the environment in the main space begins to fill up with smoke. Gray-black smoke incrementally fills the compartment for as long as the fire continues to burn, causing a reduction in visibility until a minimum visibility of five meters is reached. Once the fire is out, the smoke can be cleared by turning on ventilation fans.

2.   Steam Leak Casualty

By depressing either of the 's' or 'S' keys on the keyboard, a steam leak develops at a union on the deaerating feed tank (DFT) outlet piping just below the DFT feed isolation valve. The size of the steam leak changes if the DFT feed isolation valve is manipulated. As the valve handwheel is closed, the leak reduces in proportion to the percentage the valve is opened. Once the DFT feed isolation valve is fully shut and ventilation fans are activated, the steam can be dissipated.

The steam leak also causes the atmosphere to become obscured as in the fire casualty discussed above. The difference is that the color of the obscurity for steam is white-gray, compared to the grey-black of the fire.

APPENDIX B. EFFECT OF PVS UPON FRAME RATE

The there are two methods of measuring the effects of using PVS upon the application. The first, which most researchers report, is the number of polygons and objects which the PVS algorithm removes from culling and drawing consideration compared to the number in the entire database. The assumption is that the fewer polygons and objects left to be culled and drawn, the better the performance of the application. While this is true, a better measure of the efficiency of the algorithm is to compare frame rates with and without PVS to measure the increase in performance which PVS produces.

A. OBJECT AND POLYGON REDUCTION

The database for the entire ship contains 911 objects and 22,840 polygons. To measure the reduction in polygons in each cell, the number of objects and polygons in each cell was computed in MultiGen. The difference between these numbers and the totals for the entire model were divided by the total count to find the reduction in both object and polygons. The results are given in Table B-1.

Cell	Number of Objects	% Reduction in Objects	Number of Polygons	% Reduction in Polygons
Ops Landing	292	68.0	4676	79.5
CIC	157	82.8	2274	90.0
Radar Room	130	85.7	2218	90.3
Ops Ofice	58	93.6	571	97.5
DCC	82	91.0	961	95.8
DCC Landing	110	87.1	1180	94.8
HT Shop	98	89.2	1116	95.1
ER Landing	197	78.4	2231	90.2

Table B-1: Reduction of Polygons per Cell Using PVS

Cell	Number of Objects	% Reduction in Objects	Number of Polygons	% Reduction in Polygons
ER Upper Level	385	57.7	7644	66.5
ER Middle Level	381	58.2	7623	66.7
ER Lower Level	381	58.2	7623	66.7
Exterior	119	87.0	2464	89.2
Average	199.2	78.1	3381.7	85.2

Table B-1: Reduction of Polygons per Cell Using PVS

B. IMPROVEMENT IN FRAME RATE

1. Methodology

In order to determine the efficiency of the PVS algorithm used in the shipboard VET, an experiment was conducted which compared the performance of the PVS version of the application against a version which has had the PVS functionality removed. The methodology used was to randomly choose a point in each of the cells in the model. From there, the view was rotated a full 360 and the minimum, maximum and average frame rates were noted. The average was not simply the numerical average of the minimum and maximum, but instead was a weighted average of all the frame rates during the rotation. Since there was no method to record this data, it is an estimate by the authors, and is more inaccurate than either the minimum or maximum frame rates. However, the authors' feel that it is still fairly accurate and serves as a better indicator of the efficiency of PVS, so it is included here. The data from this experiment is included in Table B-2.

The average of all the results does not include the results from the exterior and the vehicle deck because the reduction was excessively low. This is because the frame rates were artificially high, since most of the rotation looked at the single polygon of the water.

	Reality Engine 2			Reality Engine 1			Indigo 2 Extreme		
	with PVS	w/o PVS	% red	with PVS	w/o PVS	% red	with PVS	w/o PVS	% red
Ops Lnd Low	12.0	5.5	54.2	6.7	3.5	47.8	4.8	1.6	66.7
Highest	30.0	15.0	50.0	12.0	7.5	37.5	6.5	3.5	46.1
Average	22.5	8.5	62.2	10.0	5.0	50.0	5.2	2.0	61.5
CIC Lowest	20.0	6.7	66.5	8.6	3.5	58.8	5.1	1.7	66.7
Highest	20.0	15.0	25.0	12.0	6.7	44.2	8.0	3.4	57.5
Average	20.0	10.0	50.0	10.0	5.0	50.0	6.0	2.2	63.3
Radar Rm Low	12.0	6.0	50.0	6.7	3.5	47.8	4.0	1.5	62.5
Highest	30.0	15.0	50.0	12.0	6.7	44.2	8.0	3.4	57.5
Average	22.5	9.0	60.0	10.0	5.0	50.0	6.0	2.2	63.3
Ops Ofice Low	20.0	6.0	70.0	10.0	3.8	62.0	5.0	1.2	76.0
Highest	30.0	15.0	50.0	12.0	7.5	37.5	8.0	3.8	52.5
Average	25.0	10.0	60.0	12.0	5.2	56.7	5.7	2.1	63.2
DC Central Low	12.0	5.0	58.3	6.7	3.0	55.2	4.8	1.2	75.0
Highest	20.0	15.0	25.0	12.0	7.5	37.5	7.2	3.9	45.8
Average	17.0	9.0	47.0	9.0	5.5	38.9	6.0	1.2	80.0
HT Shop Lowest	12.0	5.5	54.2	6.7	3.0	55.2	3.8	1.1	73.7
Highest	20.0	15.0	25.0	10.0	8.6	14.0	7.2	4.2	41.7
Average	16.0	10.0	37.5	8.0	5.5	31.2	5.0	1.6	68.0
DCC Landg Low	12.0	5.5	54.2	6.7	3.0	55.2	3.3	1.1	66.7
Highest	20.0	15.0	25.0	10.0	8.6	14.0	7.2	4.2	41.7
Average	17.0	9.0	47.0	8.0	4.5	43.7	4.8	1.5	67.8
ER Lndg Lowest	10.0	7.5	25.0	6.0	4.3	28.3	3.6	1.4	61.1
Highest	30.0	15.0	50.0	12.0	8.6	28.3	6.0	3.9	35.0
Average	22.0	12.0	45.5	10.0	6.0	40.0	4.8	1.9	60.4

Table B-2: Comparison of Frame Rates With and Without PVS (frames/sec)

	Reality Engine 2			Reality Engine 1			Indigo 2 Extreme		
	with PVS	w/o PVS	% red	with PVS	w/o PVS	% red	with PVS	w/o PVS	% red
ER UL Lowest	15.0	7.5	50.0	7.5	4.0	46.7	4.2	1.4	66.7
Highest	30.0	15.0	50.0	12.0	8.6	28.3	7.2	4.2	41.7
Average	22.0	14.0	36.4	10.0	5.5	45.0	6.0	2.3	61.7
ER ML Lowest	12.0	6.7	44.2	6.0	4.3	28.3	3.3	1.4	57.8
Highest	30.0	20.0	33.3	12.0	10.0	16.7	6.0	4.7	21.7
Average	20.0	12.0	40.0	9.0	6.0	33.3	5.0	2.3	54.0
ER LL Lowest	15.0	7.8	48.0	7.5	4.3	42.7	2.8	1.4	50.0
Highest	30.0	20.0	33.3	12.0	10.0	16.7	6.5	3.9	40.0
Average	23.0	15.0	34.8	10.0	7.0	30.0	5.8	2.4	58.6
Exterior Lowest	10.0	7.5	25.0	6.7	4.0	40.3	2.8	1.4	50.0
Highest	30.0	30.0	0.0	17.0	15.0	11.8	10.3	8.9	13.6
Average	22.5	20.0	11.1	9.0	6.0	33.3	7.0	5.5	21.4
Veh Deck Lowest	7.5	6.8	9.3	5.5	3.3	40.0	2.2	1.1	50.0
Highest	30.0	30.0	0.0	15.0	12.0	20.0	8.2	5.4	34.1
Average	17.0	15.0	11.8	9.0	6.0	33.3	3.5	2.1	40.0
Average Lowest	13.8	6.3	54.3	7.2	3.6	50.0	4.1	1.2	70.7
Highest	26.4	15.9	39.8	11.6	8.2	29.3	7.1	3.9	45.1
Average	20.6	10.7	48.1	9.6	5.5	42.7	5.5	2.0	63.6

Table B-2: Comparison of Frame Rates With and Without PVS (frames/sec)